,

Solver-Aided Compiler Design for Programmable Network Devices

by

Xiangyu Gao

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

September, 2024

Dr. Anirudh Sivaraman

Dr. Srinivas Narayana

Abstract

Historically, network devices were mostly fixed-function ones. They could run at a line rate of one network packet per nanosecond, but it was impossible to support newly developed network algorithms without upgrading the device. The emergence of programmable network devices remedies this drawback. These devices use reconfigurable match table model to enable programmability and provide more flexibility for developers to continue updating and adding new algorithms to the device. People developed several programming languages to write programs for these devices. Even though it is not hard to get started with writing packet-processing code, writing programs that can fit within the target devices' various resource constraints is not an easy job. The root cause for that is the lack of optimizing compilers in this domain. Hence, this thesis focuses on optimizing compiler design for domain-specific network accelerators using solver-aided techniques that can generate better compilation results compared with state-of-the-art compilers.

First, we build the Chipmunk compiler that does code generation for stateful transactions into programmable switches using program synthesis. We frame the compilation problem as a solution-searching problem and use a program synthesis engine, SKETCH, to find a semantically equivalent compilation outcome. Additionally, we also develop a series of algorithms to speed up the compilation process. We find that the Chipmunk compiler can generate better compilation results in terms of hardware resource usage within a reasonable time period.

Second, we build the CaT compiler that does both code generation and resource allocation into

the packet-processing pipeline using solver-aided technologies. We decompose the compilation problem for such pipelines into three phases—making extensive use of solver engines to simplify the development of these phases. We also incorporate some heuristics for further resource usage optimization. We observe that the CaT compiler can generate near-optimal compilation results at a much faster speed than Chipmunk.

Third, we build the Polyglotter compiler that outputs programs for target hardware devices from input programs written for source hardware devices in the parser portion. This compiler unifies features across different programming languages and reduces the efforts required to write algorithms across platforms. We discover that the Polyglotter compiler can generate correct transpilation results with better hardware resource usage.

To our best knowledge, we for the first time propose to incorporate solver-aided techniques into compiler design for programmable network devices. In the domain of programmable network devices, these compilers can outperform traditional compilers that rely on program rewrite rules. Our contributions are beyond just building solver-aided compilers and include domain-specific algorithms to speed up the whole compilation process. Based on these developed compilers, we explore several useful aspects of solver-aided techniques and hope to extend them into more applications in future works.

Acknowledgements

I was fortunate to work with two phenomenal advisors, Anirudh Sivaraman, and Srinivas Narayana, in my Ph.D. journey. I won't be who I am today without their meticulous mentorship and attention. Their profound knowledge, optimistic mental attitude, and rigorous scholarship are what I should keep learning forever.

I started working with Anirudh when I was a master student in the NYU Financial Engineering program. It was he that led me to the world of computer science. He taught me how to leverage my advantages to frame a research problem from scratch. Each time I had a technical issue, he was not only patient enough to find the solution with me but also summarized it into a lesson that I could learn in later research projects. Whenever I felt a little bit lost when doing research, Anirudh always showed the light from darkness by helping me prioritize the TODO list so that I could have clear tasks to do in the short term and avoid digressing from our long-term goal. Srinivas gave me the first Ph.D. offer in my life. Although I did not officially join his group at Rutgers University, he took me as his student and got deeply involved in all my Ph.D. research projects. He invited me to do a practice talk at Rutgers twice during my Ph.D. period, encouraged me to meet with different researchers, and emphasized understanding the audience's motivation behind their questions before giving an answer. He set himself as an example, showing the importance of being optimistic, humble, humorous, and positive as a researcher or a person. I wish I could have better performance as a Ph.D. candidate in their group. This dissertation is my way of repaying their debt.

I am grateful to the many people who helped me during the doctoral program that led to my dissertation. My collaborators played an important role in the completion of this thesis. Taegyun Kim had rich experience as an engineer at Google before his Ph.D. He did a lot of coding in the Chipmunk project and unselfishly shared several useful materials with me to improve my programming style. Pravein Govindan Kannan developed the interface to run the compiled program in the Tofino switch and thoroughly check Chipmunk's artifact evaluation process. Divya Raghunathan and Ruijie Fang spent quite a long time working overnight to debug and polish our paper writing before the submission deadline. As a quite senior faculty member, Aarti Gupta burnt her midnight oil before the ASPLOS 2023 deadline to edit our paper when we found that our paper significantly exceeded the page limits an hour before the submission deadline. Our efforts paid off. When everyone felt quite depressed about the quality of our submitted paper, an interesting "drama" happened afterward. The program chairs gave all authors a 3-day extension to polish their submission for other reasons, and we utilized this extra time to improve our paper into a much better version. Tao Wang implemented the FPGA simulator backend for the CaT project during his vacation back in China. Part of this thesis originated from my research internship at Alibaba Group and Google. Jiaqi Gao and Ennan Zhai devoted their time and energy together with me to pinpointing a meaningful research problem and sharing their view of doing practically impactful research based on their working experience at Alibaba. Hari Thantry, Shijith Chempeth, Bili Dong, and Devon Loehr familiarized me with their Intermediate Representation (IR) design for the programmable parser and showed me the strict code review process at Google. Karan Kumar Gangadhar contributed illustrative figures to our Polyglotter transpiler paper after he officially joined this project's discussion only a few times.

My other dissertation committee members helped shape my research results into a better form. Thomas Wies highlighted the importance of adding more explanations for my evaluation results in the presentation. Joseph Tassarotti explained to me the way to efficiently digest the gist of a PL-related research paper.

Other faculty members in the network and system community proposed valuable suggestions for my growth in terms of research taste and presentation skills. I met Eddy (Zheng) Zhang at ASPLOS 2023, where she gave me a comprehensive summary of her ongoing research projects and welcomed collaboration with her students (Yuwei Jin and Minghao Guo) to utilize both of our expertise. During the paper submission process, she showed me how to deliver research ideas using attractive figures and scripts by editing my writing for several passes. Yuwei and Minghao often voluntarily picked me up from the Edison train station to their office and brainstormed together to come up with better algorithms and clearer paper presentation style. I felt excited when reviewers from *Supercomputing 2024* spoke highly of our paper *Optimizing Quantum Fourier Transformation (QFT) Kernels for Modern NISQ and FT Architectures*, which showed the potential to transplant program synthesis techniques into the compiler design in other domains including state-of-the-art quantum computing architectures. Jinyang Li, my high school alumni, shared with me her opinion about being a good professor based on her academic career. Aurojit Panda was always open to me for a discussion about my research topic. His scope of knowledge and energy to attend almost all top-tier conferences and community meetings motivated me to keep working hard. Michael Walfish gave me a chance to work as a TA for his Operating System course, where he showed his strict time management style to maximize time utilization. Lakshmi Subramanian showed me the process of turning research projects into a successful start-up. Santosh Nagarakatte instructed me on how to come up with influential research ideas through critical thinking and how to advertise the research results through an interesting story. Sudarsun Kannan shared with me the experience of finding the motivation for a project and utilizing the postdoc period as a researcher. Thu Nguyen took time out of his busy schedule as a dean to meet with me, explaining technical details about optimizing electricity usage in data centers, and encouraging me to cultivate the habit of daily writing. Cheng Tan, a senior schoolmate in the NYU system lab, is the one who gave me timely help whenever I needed it. He told me how to work as a TA, how to mentor a student, and where to find useful information for Ph.D. students.

As a candidate with an economy-related background before Ph.D., I kept enhancing my knowledge base through rich course resources provided by the NYU CS department to familiarize topics outside of my research area. I want to express my gratefulness to many NYU CS faculty members, including Prof. Subhash Khot (Theory of Computation and Geometric Methods in Algorithm Design), Prof. Tang Yang (Operating System), Prof. Benjamin Goldberg (Computer System Organization), Prof. Cory Plock (Programming Languages), Prof. Yann LeCun (Deep Learning), Prof. Hasan Aljabbouli (Object Oriented Programming), Prof. Nicholas Spooner (Quantum Computing), Prof. Joseph Bonneau (Introduction to Computer Security), and Prof. Amos Bloomberg (Database Design And Implementation). Their willingness to let me unofficially participate in their lectures gives me a chance to lay a solid foundation in terms of the knowledge base. Many of my research ideas and optimizing algorithms originated from their lectures.

My lab mates past and present, Changgeng Zhao, John Westhoff, Jinkun Lin, Anqi Zhang, Ding Ding, Zhanghan Wang, Michael Dean Wong, Aatish Kishan Varma, Jessica Berg, Muhammad Haseeb, Ulysses Butler, Daniel Qian, Divyam Madaan, Tianyao Chen, Bingran Shen, Leyi Zhu, Siqi Wang, Xiaoqi Chen, Xiao Zhang, Peirui Cao, Mingyu Li, Longfei Qiu, Jiaxin Lin, Tao Ji, Jane Chen, Raj Joshi, Xiaotong Zhu, Ankit Bhardwaj, Ramakrishnan Krishnamurthy, Talal Ahmad, Shiva Radhakrishnan Iye, Lingfan Yu, Eric (Yu) Cao, Haitian Jiang, Hexu Zhao, Markus de Medeiros, Chaitanya Agarwal, Jessica (Qingyan) Chen, Kevin Choi, Arasu Arun, William Wang, Devora Chait, Nisarg Patel, Elaine Li, Jacob Salzberg, Ekanshdeep Gupta, Qiongwen Xu, Bhavana Vannarth Shobhana, Harishankar Vishwanathan, created a great environment for me to work during the day time and play poker occasionally during happy hours.

Professors and researchers from various institutes including Aditya Akella, Venkat Arun, Isil Dillig, Nate Foster, Junchen Jiang, Mina Tahmasbi Arashloo, Brent Stephens, Grace (Guyue) Liu, Chen Tian, Qiao Xiang, Sheng Xu, Nasir Memon, Cheng Zhang, Bingyang Liu, Haiping Che, Yuchun Lu, Haibo Chen, Richard Martin, Danyang Zhuo, Rachee Singh, Sun-Yuan Kung, Ratul Mahajan, Arvind Krishnamurthy, Ellen Zegura, Chen Qian, Minmei Wang, Yang Guo, Han Qiu,

Meikang Qiu, Zhaoguo Wang, Feng Qian, Zhiwei Yun, Dongming Zhu, Dongdong Ge, and Guoqiang Tian frankly offered their insightful suggestions for my career choices.

My friends and master schoolmates, Tiancheng Hou and Fan Gao, are always sources of happiness. They brought external views to my research topics, some of which contributed to the final version of the Chipmunk project. I enjoyed discussing everything meaningful (e.g., investment, sports, technology, and social affairs) with them at our weekly dinner.

Santiago Pizzini was an excellent academic administrator, assisting with all logistics and always responding to my emails on time.

My family members always provided endless support for me when I went through both highs and lows. My parents, Yang Gao and Guang Yang, cultivated me to become a person with presence of mind in the face of important events. My grandparents have reminded me to aim high and never give up easily since my childhood. My grandfather, Jihou Yang, and his brother, Prof. Jichu Yang, motivated me to take the challenge of a Ph.D. position and do something useful for the world when I had doubts about myself before my Ph.D. application. My three-year-old daughter, Chenyun Gao, is the best prescription to eliminate my fatigue and worry. My depression disappeared immediately after looking at her cherubic face. My in-laws, Pingfan Liu and Suhong Zhou, unconditionally undertake childcare tasks, enabling me to spend more time making progress in my research projects. My wife, Jiazhen Wei, voluntarily sacrificed her previous job in Shanghai and moved to another continent to face various challenges together with me in the United States.

I dedicate this dissertation to anyone who makes direct or indirect contributions to my growth.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Background

Programmable network devices have become more and more popular due to the flexibility they can provide. This flexibility allows developers to implement customized network functions (e.g., load balancing [128], network monitoring [45]) into these devices. A series of programming languages [31] [99] [130] [46] [60] [137] are developed to enable developers to write programs for these devices. It is easy to get started with writing packet-processing programs by reading the tutorial or attending a workshop, but ensuring that the written program can obtain a successful compilation is quite hard. This requires developers to have a deep understanding of both the target devices' hardware constraints and the programming languages' syntax.

This thesis focuses on optimizing compiler design for programmable network devices to alleviate the difficulties for developers in writing fast packet-processing programs.

1.1.1 Programmable network devices

Traditional network devices are mostly fixed-functional. They can support several less complex network functions (e.g., packet forwarding [119], network flooding [81]). Typical functionalities include packet forwarding from one switch to another one according to rules within the routing table. However, with the development of various network technologies, network developers have motivations to implement more complicated algorithms (network measurement [63] [156],

load balancing [125], and firewall detection [19]) into network devices. Programmable network devices started to emerge. Typical devices include Barefoot Tofino [114], Broadcom Trident 4 [148], Pensando SmartNIC [14], and Intel IPU [69]. They not only provide flexibility for developers to frequently update their programs to implement but also guarantee their performance (e.g., throughput and latency).

1.1.2 Programming Language design

A series of programming languages have been designed from both academic and industrial fields to enable developers to write programs for these programmable network devices. P4 [104] and NPL [99] are widely used by many cloud providers including Alibaba, Google, Intel, and Broadcom.

A series of new languages emerge to remedy the shortcomings of P4 and NPL. These languages are designed to offer more convenience to program developers. For example, Domino [130] proposes the packet transactional semantics; this feature was later integrated into P4-16 as the atomic function. P4All [60] extends P4 by defining elastic data structures that stretch automatically to make optimal use of available switch resources; Lucid [135] introduces abstractions to allow sophisticated data-plane applications with interleaved packet-handling and control logic; Lyra [46] proposes a cross-platform high-level language that allows programmers to use simple statements to express their intents. These efforts to develop new programming languages make the programmers utilize the hardware device's expressiveness conveniently.

These programming language designs are based on the hardware device's architecture configuration. As an example, languages such as P4 use pragma [103] to support concurrent execution for multiple tables. Programs written by these languages consists of multiple modules (e.g., parser, pipeline, deparser) in sequence, each of which is mapped to one part of the target hardware devices. Compared with other general-purpose languages (e.g., C, C++, and Java), these network programming languages are less expressive. Specifically, these languages do not sup-

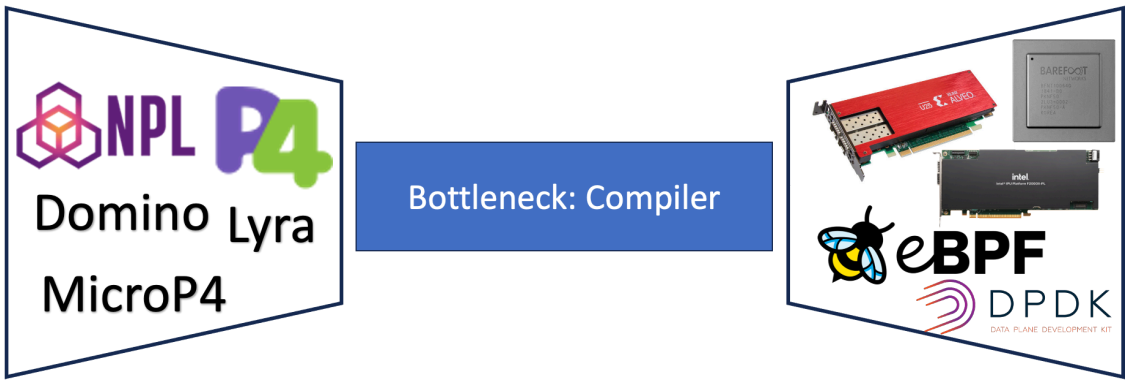

Figure 1.1: Compiler becomes the bottleneck to use programmable network devices.

port advanced data types such as pointers or allow unbounded loops and recursion operations. Therefore, this also restricts the complexity of programs that can run over the network devices.

1.1.3 Lack of optimizing compiler

Figure 1.1 visualizes the current trend in the domain of programmable network devices. On the one side, chip vendors produce several hardware devices to offer developers multiple choices; on the other side, programming languages are developed for engineers to write programs for these devices. In order to execute the developed code into a real hardware device, we need good compilers to efficiently turn the input program into executable code within the target hardware resources. Unfortunately, there is a lack of good compiler design in this domain, making the compiler become a real bottleneck for further development of programmable network devices.

Current compilers used in this area might output different compilation results (success vs. failure) for semantically equivalent programs. The reason is due to that compilers are unable to fully explore the search space of machine code programs that could implement the high-level program. As an example, the Domino compiler has a series of rules to rewrite [42] the input program to a format that will be mapped to ALUs of programmable switch; but these rewrite rules can unnecessarily turn some input program into a format that is hard to map to the target hardware device. Therefore, it could either output compilation results with suboptimal resource

usage or reject the compilation even though there is a way to express the input program's semantic. Using more than available resources, a potential result from suboptimal resource usage, might lead to a compilation failure as well.

Such a case appears not only in academic compilers [130] but also in commercial compiles [102]. This forces programmer developers to write their program in a "compiler-friendly" way so that the compiler can accept the input program by generating the compilation result. This is quite hard or even impossible because developers either have to understand the compiler's code base in detail if it is open-source or guess its working mechanism if it is close-source.

This thesis aims to bridge the gap between these programming languages and programmable network devices by developing optimizing compilers. An ideal compiler should always output the correct compilation outcome regardless of the written style of input programs. Existing compilers involve several program rewrite rules designed by the human's heuristic. This might lead to differences in compilation output on semantically equivalent input programs. Depending on how much hardware resources (e.g., pipeline stages, match-action tables) each compilation output consumes, such difference could finally lead to a false positive compilation rejection when it requires more than available resources. This work makes a proposal: leveraging solvers to exhaustively search all possible solutions to find a semantically equivalent one within the search space. Such a proposal makes the final output irrelevant to the written style of the input program and avoids suboptimal hardware resource usage. The efforts to do optimizing compiler design can be complementary to the current work on language design and hardware design.

1.2 Primary contributions

We summarize three main contributions of the thesis in this section.

1.2.1 Rethinking the compilation process for programmable network devices

Compilers are software systems that translate human-written programs into a form in which a device can execute it. Traditional compiler design [9] involves program-rewrite rules and target-dependent optimizations to improve the compilation outcome. Compiler developers execute these rules on the basis of their understanding of the input programming language syntax and the target device's hardware features. However, such a way to develop compilers for programmable network devices might cause problems. As an example, some rewrite-rules in the Domino [130] compiler require unnecessary complex expressiveness for stateful ALU; the order to implement these rewrite-rules [100] may also affect the compilation outcome. These may lead to potential problems where developed compilers output incorrect compilation results or outcome with suboptimal hardware resource usage. This thesis aims to remedy these compilation problems by viewing the compilation problem from a novel angle: leveraging solver-aided techniques to optimize compiler design. As long as we encode the capability of the target hardware device, no further rewrite-rules are needed because the solver can help us find a solution within the search space.

Specifically, we encode compilation into a space-searching problem and leverage various solvers to find a semantically equivalent solution from the search space. Instead of missing a solution or finding a solution with sub-optimal resource usage by using traditional compilers, solver-aided compilers can avoid missing any solutions by exhaustively searching all possibilities within the defined search space. This contribution successfully explores the synergy effect between solver-aided techniques and compiler design.

1.2.2 Domain-specific speeding up algorithms

Developing solver-aided compilers involves encoding all types of constraints (e.g., computation ability and hardware resources) in a way that can be interpreted by solvers. This process is not straightforward and Chapter 3, chapter 4, chapter 5 describe detailed encoding methods. It is not sufficient even if we successfully frame the compilation as a solution-searching problem. One of the most important bottlenecks to using solvers is the long running time especially when compiling complex input program to hardware devices with complicated configurations. So we need to develop several domain-specific algorithms to expedite the compilation process.

We utilize the unique features for programmable network devices and incorporate several speeding-up algorithms into the compiler design including *slicing* [50], *constant synthesis* [50], *phase division* [48], *granularity* [49]. At a high level, these algorithms aim at decomposing the problem into multiple sub-problems to reduce the size of the search space. Then, we can get the final results by combining solutions from all sub-problems. Such decomposition can achieve significant improvement in the compilation speed.

1.2.3 Retargetablity to newly emerging devices

When developing the solver-aided compiler, we do not want our design to be restricted to only one type of device. Instead, the goal is to make the developed compiler retargetable so we need to make only a few changes to the existing compiler for a new hardware device.

This thesis starts from a compiler design for the packet-processing pipeline of programmable switches such as the Barefoot Tofino switch [114] and an FPGA simulator [150]. Later, we extend the compiler design into the parser part of several target devices (e.g., Barefoot Tofino, Broadcom Trident 4 [148]). Our compiler design might permit rapid prototyping of compilers. This is because synthesis allows us to declaratively specify code generation for different substrates as space-searching problems. This could allow us to reuse solver-aided techniques for performing

code generation across different devices.

Two major parts will determine the hardware device's capability: (1) the hardware configuration and (2) the expressiveness of computation. These two parts are flexible to change in our compiler design. As for the hardware configuration, we capture the hardware architecture, model its available hardware resources, and restrict using more than available resources; as for the computation expressiveness, we use a search space to represent all possible computations offered by the target devices. This mainly consists of the expressiveness within each arithmetic logic unit (ALU) and the input/output connection across ALUs. Different ALUs (e.g., if-then-else, read-and-write) have different expressiveness.

All hardware resource (e.g., # match-action tables, # ALUs, and memory size) limits are parameterized, and their final value is device dependent. In our design, we can realize the retargetability by changing the hardware configuration, updating the ALU's expressiveness, and modifying the hardware resources' parameters.

1.3 Program Synthesis

Program synthesis is the process of automatically generating a program that meets a given specification. This thesis uses program synthesis as a subroutine in the compilation process to do solution exploration (synthesis) and check semantic equivalence (verification). The program synthesis engine used in this dissertation is SKETCH [132].

1.3.1 SKETCH examples

We focus on a recent variant of program synthesis called *syntax-guided synthesis (SyGuS)* [12, 132, 147] that constrains the search space of programs using syntactic restrictions. As a concrete example of syntax-guided synthesis, in SKETCH [131, 134], a programmer provides a program synthesizer with the specification along with a *sketch* (Figure 1.2): a partial program with *holes*

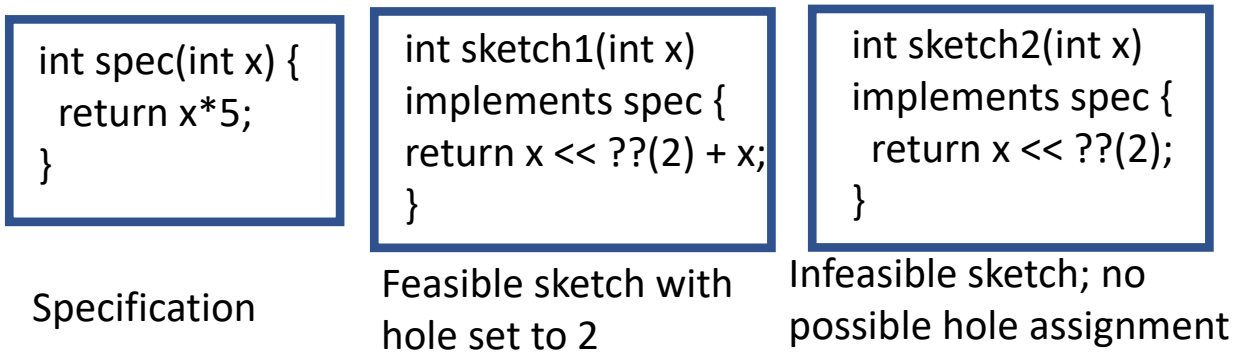

Figure 1.2: Syntax-guided synthesis in SKETCH. ??(b) is a hole whose value is in $[0, 2^b - 1]$. « is the left-shift operator. In this case, ??(2) represents a constant in $\{0, 1, 2, 3\}$.

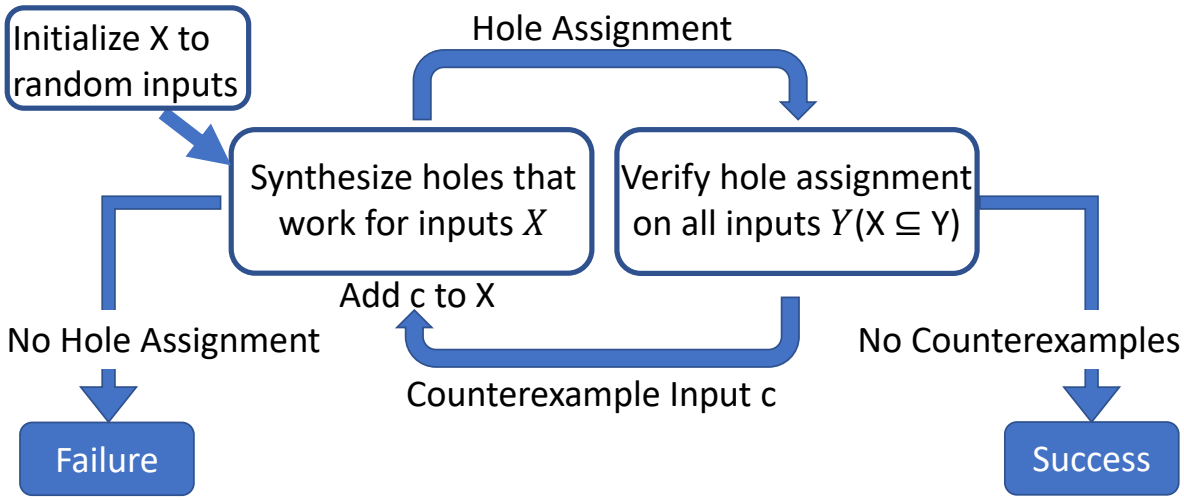

Figure 1.3: The CEGIS algorithm for synthesis.

representing values within a finite range of integers. The partial program constrains the search space syntactically and encodes the programmer's insight into the structure of the implementation. The synthesizer completes the sketch by filling in all holes with concrete values so that the completed sketch meets the specification, or says that synthesis is infeasible.

1.3.2 The CEGIS loop

We briefly describe SKETCH's [132] internals here. SKETCH is given as input a specification to satisfy and a partial program (the sketch) (Figure 3.4). Let x be an n-bit vector representing

all inputs to both the specification S and the partial program P. The task of the synthesizer is to determine the values of all the holes in P such that the results of executing the specification and the sketch on an input x, $S(x)$ and $P(x)$, are the same for all x. Let c be an m-bit vector representing all holes that need to be determined (or "filled in") by SKETCH to complete the sketch. Then, the program synthesis problem solves for c in the following formula in first-order logic [134]:

$$\exists c \in \{0,1\}^m, \forall x \in \{0,1\}^n : S(x) = P(x,c) \tag{1.1}$$

Equation 1.1 is an instance of the quantified boolean formula problem (QBF) [116]. QBF is a generalization of boolean satisfiability (SAT) that allows multiple $\forall$ and $\exists$ quantifiers; SAT implicitly supports a single $\forall$ or $\exists$. While QBF solvers exist [40, 90], they are not optimized for the QBF instances found in program synthesis [132]. Hence, SKETCH uses an algorithm called *counterexample-guided inductive synthesis (CEGIS) [133, 134]*, designed to work efficiently for the QBF instances found in program synthesis.

CEGIS (Figure 1.3) exploits the *bounded observation hypothesis*: for typical specifications, there are a small number of representative inputs that form a "perfect test suite," i.e., if the specification and the completed sketch agree on this test suite, then they agree on all inputs. To exploit this hypothesis, CEGIS repeatedly alternates between two phases: (1) synthesizing on a small set of concrete test inputs and (2) verifying that the completed sketch matches the specification on all possible inputs. A failed verification generates a counterexample that is added to the set of concrete test inputs, and a fresh iteration of synthesis+verification follows. CEGIS terminates when either the verification phase succeeds or the synthesis phase fails, i.e., there is no way to find values for the holes that allow P and S to match on the concrete test input set.

The synthesis phase of CEGIS is represented by the following formula. Here $x_1, x_2, \ldots x_k$ are the current set of concrete test inputs:

$$\exists c \in \{0,1\}^m : S(x_1) = P(x_1,c) \wedge \ldots S(x_k) = P(x_k,c) \tag{1.2}$$

The verification phase is represented by the following formula. Here, c^* is the hole solution being verified:

$$\forall x \in \{0,1\}^n : S(x) = P(x, c^*) \tag{1.3}$$

Both the synthesis and verification phases of CEGIS are simpler than solving Equation 1.1 directly as a QBF problem. This is because each phase fixes either the test inputs (synthesis) or holes (verification) to concrete values, which turns the resulting subproblem into a SAT problem, which can be fed to a more efficient SAT (instead of QBF) solver.

1.3.3 More applications of program synthesis

There have been many promising real-world applications of synthesis [6, 76, 110, 123, 127]. Recent network researchers leverage program synthesis tools to develop efficient congestion control rules [5], analyze network performance [18], and develop compilers to output high-quality BPF bytecode [154] or qubit circuits for quantum-computing architectures [73].

Additionally, the syntax-guided synthesis competition encourages developers to continue improving the program synthesis speed by testing various solvers for syntax-guided synthesis problems on a large collection of benchmarks. Faster program synthesis speed can broaden the application scenario for this technique.

Therefore, in this thesis, we want to show that *program synthesis can be used as an enabling technology for the compiler design for programmable network devices.*

1.4 Packet-processing network devices

At a high level, a packet-processing network device [114, 148] consists of several major parts: parser, packet-proessing pipeline, and deparser. The functionality of a parser is to do preparation

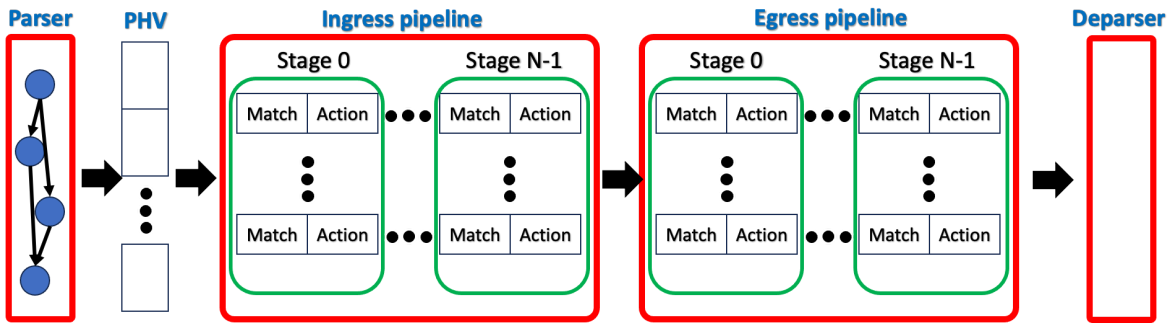

Figure 1.4: Programmable network devices architecture.

by identifying all necessary information for the packet-processing pipeline to process. It takes as an input a bit stream and outputs a vector containing the initial values of all packet fields required for the pipeline. Then, the packet-processing pipeline does packet modification by updating persistent switch states and values stored in the packet header vector (PHV). The switch states are visible to all subsequent packets while PHV stores values that are specific to one packet. One persistent switch state is only accessible by one and only one ALU. Finally, the deparser re-assembles PHV containers to a packet.

The general architecture of a programmable network device is shown in Figure 1.4. The remaining of this section provides further descriptions for both the parser part and the packet-processing pipeline part.

1.4.1 Programmable parser

At a high level, a parser identifies headers and extracts packet fields for subsequent packet processing. The whole process can be regarded as a finite state machine (FSM), consisting of several parser states and transition rules. Kangaroo [80] and Gibb *et al.* [51] discuss several principles and dynamic programming algorithms for the parser design.

There are 2 types of parsers: the fixed-function parser and the programmable parser depending on whether the device provides developers the flexibility to design customized parsing logics.

This thesis focuses on the compilation of the programmable parser.

Functionality of a parser. The fundamental functionality of a parser is to take as an input a bit stream and output a vector of packet headers which contains the initial value of all packet fields. It uses one FSM to determine the header field extraction logic. Each state within the FSM is called a *parser node* which extracts from the input bit stream and assigns value to corresponding packet fields. The state transition rules, called *transition logic*, within a parser decide the next parser node that a parser node should transit to based on the values of several variables (e.g., packet fields and temporary variables).

Heterogenuous parser behavior. The concrete parser behaviors are different across hardware devices. For example, the Broadcom Trident 4 switch [148] allows interleaving the parser and the pipeline while the Barefoot Tofino switch [114] disallows this behavior. Specifically, we can use *parse break* [99] to jump out of the parser and do table operations to update some variables. Then, we can return to the parser after *parse continue* where those updated variables are visible and required to determine the subsequent parsing behavior. Even though the Barefoot Tofino switch cannot allow such interaction between parser and pipeline, it is possible to express the same semantics by using the data structure *valueset* [104]. Developers use a valueset to check whether or not a value belongs to a set. This behavior can simulate the match action table's operation to some extent.

Chapter 5 in this thesis builds a transpiler, Polyglotter, to realize cross-platform parser code generation using program synthesis. This transpiler encoded features from different hardware devices to ensure that the output program follows the target device's hardware constraints.

1.4.2 Reconfigurable match table pipeline

The parser generates a packet header vector (PHV) containing values of all packet fields required for the reconfigurable match table (RMT) [30] pipeline. The packet-processing pipeline implements the program developer's algorithm and updates corresponding variables.

Developers use 2 types of variables to determine the final network function. They use stateless variables (e.g., packet fields and temporary variables) to represent anything that is **packet-specific** while they use stateful variables to represent anything that is **device-specific**. Only the updated values of stateful variables are visible by the subsequent packet trace. This RMT pipeline consists of several match action tables laying in an $m * n$ grid. Each table does variable modification based on particular match-action rules decided by the control plane. All match action tables within the same stage execute concurrently. Tables placed in later stages will not execute until tables in previous stages complete their execution. PHV containers are used as the intermediate to pass values across tables from different stages. This pipeline is reconfigurable, meaning the functionality of a programmable device is not fixed because there is some room to try different algorithms to run over the programmable device.

Chapter 3 and chapter 4 leverage multiple solver-aided techniques to build compilers (e.g., Chipmunk [50] and CaT [48]) for the packet-processing pipeline.

1.5 Uniqueness in compiler design for programmable network devices

At a high level, a compiler is to turn one source program written by a high-level language such as C language into one target program written by a low-level language such as machine code to execute over a target hardware. The compiler developers should understand the syntax and semantics of the source and target languages to guarantee that the output program can precisely execute what programmers want to run over this machine. There have been already a lot of works [9] describing the general process of compiler design. However, several unique features in programmable network devices make the compiler design a special problem in its own right. There are at least 4 ways in which packet processing is different and we will describe them one by one in detail.

Narrower domain than general-purpose hardware. First, it's a narrower domain. There are DSLs for packet processing such as P4-14 [144], P4-16 [104], and NPL [99]. Their language designs are restricted subsets of C. This kind of narrowness gives us leverage: allowing us to use new techniques or revisit old techniques that may not be applicable to compilers like GCC. For instance, these network programming languages lack pointers data structure and do not support operations such as loop and recursion, making several compiler problems both practically and theoretically easier.

Common features across programmable network devices. Second, many platforms share a common feature of computing under resource constraints. For instance, on a hardware switch/NIC pipeline, we can typically access a piece of memory exactly once as the packet goes through the pipeline; stateful variables such as persistent switch states are accessible only by one and only one ALU; there is also a limited amount of memory resources (e.g., TCAM and SRAM) developers can use. Therefore, these commonalities make it possible to build a compiler that can be easily extended to use across all these network devices.

Importance of performance improvement. Third, there is a significant value attached to high performance in these settings—whether that is hitting line rates of 100Gbit/s or achieving latencies in the microseconds. This originates from the *all-or-nothing* feature of programmable network devices. A compiler can either successfully compile an input program to run at the full line-rate of the target devices or nothing could run at all. There is no middle ground in between. It could be valuable to spend more effort required to get the high throughput performance of the compilation result.

Less frequent change for network program. Last but not the least, packet-processing programs change relatively infrequently compared with programs written for other platforms (e.g., CPU, GPU). This is a luxury that other compilers like Clang and GCC do not have.

In particular, several network programs are compiled once and used forever. Hence, a good compilation result is essential. It is usually affordable for programmers to patiently wait for a

longer time in the compilation process in order to explore a high-performance implementation.

1.6 Lessons learned

We conclude this chapter by summarizing general lessons for compiler design using solver-aided techniques that go beyond the specific contributions of the 3 projects mentioned above.

1.6.1 Incoporating solvers into compiler design

Fundamentally, the compilation is to find a semantically equivalent representation that can run in the target hardware device. In the domain of programmable network accelerators, compilation involves translation from the source language to the target language in a semantically preserving way and mapping the target program to each type of hardware resource. Therefore, we can use solvers to guarantee the semantic equivalence and explore a good resource allocation result. Such a compilation approach can avoid missing any solutions by exhaustively trying all possibilities within the search space.

Given the domain of programmable network devices is relatively niche, meaning both the input program and the hardware architecture are relatively simple compared with general-purpose programs and devices, it is beneficial for us to leverage solver-based techniques to explore a better compilation outcome than traditional rule-based compilers. We believe that these solver-based techniques should not be restricted to the compiler design for programmable switches and can be extended to other network devices (e.g., IPU, DPU, SmartNIC, and FPGA) or even software platforms (e.g., eBPF, DPDK framework).

1.6.2 Human intelligence to speed up compilation

Running time is the main Achilles Heel of using solver-based techniques. The search space of candidate programs might be exploding exponentially as the complexity increases. Naively

encoding the compilation problem into a solution searching problem for a solver might always cause the forever compilation time. Therefore, it is necessary to leverage human intelligence to come up with domain-specific speeding-up algorithms.

All of the 3 projects in this thesis contain speeding-up algorithms. Chapter 3 developed algorithms such as *slicing* to speed up program synthesis solver; Chapter 4 decouples stateful computations apart from stateless computations to simplify the synthesis query; Chapter 5 does code generation for state transition logics within *ONE* parser node instead of the whole parser to reduce the search space.

All these algorithms originated from human's understanding of the input program and target hardware devices. Therefore, even though the search space for solvers might be very large, developers are always encouraged to exploit human intelligence to help the solver prune its search space and reduce the time required to find a final solution.

1.7 Source code availability

The source code for the systems presented in this dissertation is available online. For Chipmunk, the source code is given in `https://github.com/chipmunk-project` where detailed instructions to reproduce the result are in `https://github.com/chipmunk-project/chipmunk-project.github.io/blob/master/instructions.pdf`; for CaT, the source code is in `https://github.com/CaT-mindepth` together with the detailed artifact evaluation steps in the Appendix of CaT paper [48].

2 | Previously Published Material

Chapter 2 combines materials from two previous publications [47] [50]:

- Xiangyu Gao, Taegyun Kim, Aatish Kishan Varma, Anirudh Sivaraman, and Srinivas Narayana. Autogenerating Fast Packet-Processing Code Using Program Synthesis. In *HotNets*, Princeton, U.S.A, November 2019.

- Xiangyu Gao, Taegyun Kim, Michael Dean Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. Switch Code Generation Using Program Synthesis. In *SIGCOMM*, Virtual, August 2020.

Chapter 3 revises a previous publication [48]: Xiangyu Gao, Divya Raghunathan, Ruijie Fang, Tao Wang, Xiaotong Zhu, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. CaT: A Solver-Aided Compiler for Packet-Processing Pipelines. In *ASPLOS*, Vancouver, Canada, March 2023.[1] [2]

Chapter 4 extends a previous publication [49]: Xiangyu Gao, Jiaqi Gao, Karan Kumar Gangadha, Ennan Zhai, Srinivas Narayana, and Anirudh Sivaraman. Cross-Platform Transpilation of Packet-Processing Programs using Program Synthesis. In *APNET*, Sydney, Australia, August 2024.

[1] Credit to **Ruijie Fang** for his proposal of the guarded dependency analysis in the transformation phase.
[2] Credit to **Divya Raghunathan** for her developing the synthesis procedure in the synthesis phase.

3 | Switch Code Generation Using Program Synthesis

Writing packet-processing programs for programmable switch pipelines is challenging because of their *all-or-nothing* nature: a program either runs at line rate if it can fit within pipeline resources, or does not run at all. It is the compiler's responsibility to fit programs into pipeline resources. However, existing switch compilers, which use rewrite rules to generate switch machine code, often reject programs because the rules fail to transform programs into a form that can be mapped to a pipeline's limited resources—even if a mapping actually exists.

This paper presents a compiler, Chipmunk, which formulates code generation as a program synthesis problem. Chipmunk uses a program synthesis engine, SKETCH, to transform high-level programs down to switch machine code. However, naively formulating code generation as program synthesis can lead to long compile times. Hence, we develop a new domain-specific synthesis technique, *slicing*, which achieves 1–387× and 51× speed-up on average in terms of the compilation time.

Using a switch hardware simulator, we show that Chipmunk compiles many programs that a previous rule-based compiler, Domino, rejects. Chipmunk also produces machine code with fewer pipeline stages than Domino. A Chipmunk backend for the Tofino programmable switch shows that program synthesis can produce machine code for high-speed switches.

3.1 Introduction

There has been a recent flurry of research on programming languages and hardware designs for high-speed programmable switch pipelines [30, 31, 34, 66, 99, 114, 136]. Today, it is possible to specify packet processing for line-rate switches at a high level of abstraction using languages like P4, making it easy for researchers, network operators, and equipment vendors to start programming switches.

However, writing *optimized* programs for line-rate switches is much more challenging than sample programs and tutorials [103] might suggest. A realistic switch program [71, 72, 106] must fit within highly constrained switch resource budgets to run successfully. Examples of resources include pipeline stages, arithmetic logic units (ALUs), SRAM memory for control plane rules, and containers for packet headers. To make things worse, packet-processing pipelines have an *all-or-nothing* characteristic: programs that can be accommodated within the switch's resources run at the line rate of the switch pipeline; otherwise they cannot run at all. Unlike processors, there is no middle ground where complex programs can run with slightly degraded performance. This forces developers to grapple with low-level details of the hardware such as the configurations of the available ALUs, sequencing of stages, and the usage of the available stage memory (both SRAM and TCAM), to squeeze their programs into the pipeline's resources.

The difficulty of writing optimized programs can be addressed using compilers. Today's switch compilers [100, 130] are structured around rewrite rules [8] that operate on small program fragments at a time. These rules repeatedly transform the program into simpler forms until it can be easily mapped to machine code. However, rule-based compilers can spuriously *reject* many programs as they are unable to rewrite them to a form that can fit within the limited switch resources, even when there exist ways to fit those programs into the switch. §3.3 provides an example.

Motivated by these drawbacks of rule-based compilers and inspired by the success of program

synthesis in other domains [44, 55, 111, 112, 123], Gao et al. [47] observed that we can leverage *program synthesis,* i.e., automatically generating program implementations that satisfy a specification, to produce fast packet-processing code that fits within resource limits. The workshop paper observed that synthesis can be used to transform a high-level program (e.g., in C, P4-16 [104], or Domino [130]) into low-level machine code (e.g., ALU opcodes in a switch pipeline) by treating the machine code as the program to be synthesized and the high-level program as the specification. The current paper builds on the vision in that workshop paper and makes two main research contributions in designing a switch compiler, Chipmunk.

Domain-specific synthesis techniques (§3.4). Synthesis is a combinatorial search problem over a space of implementations that may satisfy a specification. Hence, a synthesis-based compiler can take much longer to generate code than a rule-based compiler. We developed a new technique, *slicing*, to speed up synthesis-based compilation for pipelines. In slicing, we decompose the synthesis problem for switch pipeline code generation into a collection of independently synthesizable sub-problems called slices. In each slice, Chipmunk synthesizes a sub-implementation that has the same behavior as the specification, but just on a *single* packet field or state variable from the specification. These sub-implementations can be directly "stacked" on top of each other to form the full pipelined implementation. Each slice presents a simpler synthesis problem—since the implementation must only respect the specification for one variable—and hence can be synthesized with fewer pipeline stages and ALUs than the original specification. The reduction in stages and ALUs leads to a much smaller search space and time for synthesis. Beyond slicing, we also adapted several techniques from program synthesis to the context of packet processing (§3.4.5).

Retargetable code generation using a pipeline description language (§3.5). We designed the Chipmunk compiler to target two backends, a switch pipeline simulator and Barefoot's Tofino switch [114], as well as subsets of their full capabilities. Our experience made it clear to us that it would be useful to share the same underlying program synthesis technology across several back-

end targets. That is, our compiler should be *retargetable*: it should be able to generate machine code for different switch pipelines with different instruction sets driven solely by a declarative specification of the hardware's capabilities [39, 85]. To enable this, we developed a declarative domain-specific language (DSL) to specify the capabilities of a pipeline's ALUs and the interconnect between them. We call this DSL the *pipeline description language.* Chipmunk takes a description of the hardware written in this language, automatically formulates a synthesis problem for an off-the-shelf synthesis engine, SKETCH [134], solves it, and translates the results of synthesis into backend-specific machine code. For the Tofino backend, which doesn't support direct programming in assembly language, we used compiler pragmas to gain the low-level control over hardware resources required for code generation (§3.6).

We evaluated Chipmunk on both the simulator and Tofino backends using 14 benchmarks drawn from a variety of sources [77, 98, 130]. Our primary findings are that:

- Chipmunk successfully compiles many programs that a rule-based compiler, Domino [130], rejects.
- Chipmunk produces machine code with fewer pipeline stages—a highly constrained switch resource—relative to Domino.
- Slicing speeds up synthesis by 1–387× (average: 51×).
- Although Chipmunk is slower than Domino, Chipmunk's compile times are within 5 minutes on 12 out of 14 benchmarks, and within 2 hours for the rest.
- Chipmunk generates Tofino machine code for 10 out of 14 benchmarks. We believe the other 4 benchmarks are beyond the capabilities of Tofino ALUs (§3.7.2).

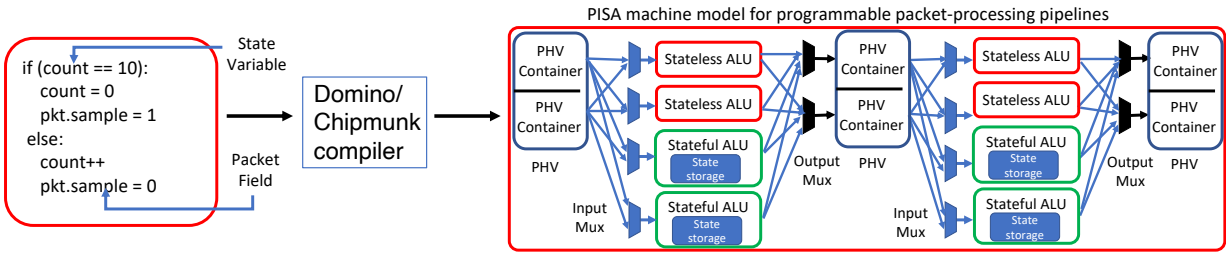

Figure 3.1: Program as a packet transaction in Domino along with a 2-by-2 pipelined grid (i.e., 2 stages, 2 stateful + 2 stateless ALUs per stage) showing the PISA machine model. Input muxes are used to determine which PHV container to use as an ALU operand. Output muxes are used to determine which ALU to use to update a container.

3.2 Background

Programming languages for packet processing. Several languages now exist for packet processing, e.g., P4-14 [144], P4-16 [104], POF [136], and Domino [130]. This paper uses Domino as the language in which the input program is specified by the programmer. Domino is well-suited to expressing packet processing with an algorithmic flavor, e.g., maintaining sketches for measurement or implementing the RCP [142] protocol. Figure 3.1 shows an example Domino program that samples every 11^{th} packet going through a pipeline. Domino provides transactional semantics: operations in a Domino program execute from start to finish atomically, as though packets are being processed by the target exactly one packet at a time. This frees the programmer from having to deal with concurrency issues, delegating that to the compiler instead. The same transactional semantics are also supported by P4-16's `@atomic` construct [105]. P4-16's `@atomic` construct was influenced by Domino [35]; hence, we expect to also be able to support P4-16 `@atomic` in the frontend.

Packet-processing pipelines. A programmable switch consists of a programmable parser to parse packet headers, one or more programmable match-action ingress pipelines to manipulate headers, a packet scheduler, and one or more programmable match-action egress pipelines for additional header manipulations. We focus on the pipelines because that is where packet

manipulations primarily occur. We consider a pipeline architecture for packet-processing based on RMT [30] and Banzai [2], which extends RMT with stateful computation. This architecture is now commonly known as the Protocol Independent Switch Architecture (PISA) [115] and is seen in many high-speed programmable switches [59, 66, 93, 114].

In PISA, an incoming packet first enters the parser. After parsing, the parsed packet headers are deposited in a *packet header vector (PHV):* a vector of containers each of which stores a single header field (e.g., IP TTL). This PHV is passed through the ingress and egress pipelines. Each pipeline stage contains multiple match-action tables that operate concurrently on PHV containers. Each match-action table identifies the rule of interest for the current packet using the *match unit* (e.g., SSH packets can be matched using a rule specifying TCP port 22) and modifies the packet using the *action unit* (e.g., adding 1 to a packet field) tied to that rule. The pipeline can maintain a small amount of action-unit-local *state* to perform the action, e.g., maintain a count of all SSH packets.

The PISA pipeline is assumed to be *feed-forward:* packets can only flow from an earlier stage to a later one, but not in reverse. This means that computations in a later stage can depend on computations in earlier ones, but not vice versa. In particular, a piece of state stored in a pipeline stage can be read, modified, and written *only once* by a packet as it passes through the pipeline. Switches can recirculate packets back into the pipeline to enable backward flow, but recirculation greatly degrades packet-processing throughput and we do not consider it here.

We refer to the action units in packet-processing pipelines as **Arithmetic Logic Units (ALUs)**. In this paper, we only focus on the pipeline's ALUs (not the match units) because the ALUs are where per-packet computation occurs—and hence the target of code generation. We further assume that the ALUs execute on all packets going through the pipeline. It is straightforward to implement use cases where the ALUs only execute on a subset of the packets because match rules can be added to the corresponding match-action tables to support such use cases. Hence, the entire pipeline can be abstracted out as a 2D grid of ALUs (Figure 3.1).

ALUs must process packets at line rate. Hence, an ALU should be able to process a new packet every clock cycle (~1 ns). ALUs can be *stateless,* i.e., operating only on PHV containers; or *stateful,* i.e., operating both on ALU-local state and PHV containers. For stateless ALUs, the ALU should be able to update a new PHV container every cycle. For stateful ALUs, the entire read-modify-write operation on the state that the ALU operates on must complete within a cycle. This guarantees state consistency even if packets in consecutive cycles access the same state. Each ALU has a set of input multiplexers (*muxes*), one per operand. These muxes are used to determine which PHV containers are used as ALU operands. Each ALU provides certain operations (e.g., addition) and may take immediate operands. Each PHV container is fed by an output mux to determine which stateful/stateless ALU's output updates it.

Compiling programs to pipelines. A compiler for a pipeline (e.g., Domino [130]) takes a packet-processing program, written in a high-level language, and turns it into low-level machine code representing pipeline configurations, e.g., ALU opcodes, allocation of packet fields to PHV containers, and configurations of muxes (Figure 3.1).

Compiling programs to pipelines is *all-or-nothing*: successfully compiled programs can run at the pipeline's line rate, but a program that is rejected by the compiler can't run at all. Programs can be rejected for two reasons. The first reason is violating resource limits: the machine code generated by the compiler might consume more resources (e.g., stages, ALUs, rule/table memory) than available. The second reason is violating computational limits: the compiler might be unable to find a way to map computations in the program to the hardware's ALUs, even with infinite ALUs.

This places a significant responsibility on the compiler, which should ideally be able to find *some* machine code corresponding to the given high-level program—provided the program is within the resource and computational limits of the pipeline: the program's computations belong to the finite space of computations possible using a single pass through the pipeline's ALUs without recirculation. As an example of a program that exceeds these limits, if the pipeline only

supports increment operations on state (but no multiply), and the program requires an exponentially weighted moving average filter over queueing delays, it is impossible to run the program using the pipeline.

3.3 The Case for Program Synthesis

Drawbacks of rule-based compilers. Compilers for packet-processing pipelines often reject programs spuriously: a semantically-equivalent version of the same program will be accepted by the same compiler. We have observed this problem with both commercial [102] and academic compilers [42].

We illustrate the problem of spurious program rejections in the context of the Domino compiler [130] using a simple example. While this example is simplified for illustration, we have observed similar spurious program rejections with more complex examples as well. Figure 3.2 shows two Domino programs written to target a PISA pipeline. The two programs are semantically equivalent, i.e., given the same input packets and the same initial state variables, they will both produce the same trace of output packets and state values at run time. However, the Domino compiler exhibits a butterfly effect or a false positive compilation result: it successfully compiles the first program, but rejects the second one even though both of them have the same semantics.

To understand why, we look at the intermediate representation (IR) constructed by the Domino compiler. This representation is akin to a directed acyclic graph (DAG) of computations, but additionally groups together stateful computations that must finish atomically within one clock cycle. Figure 3.3 shows the DAGs for both versions of the program. The shaded nodes show stateful computations, while the unshaded nodes show stateless computations [130]. The DAGs differ in the complexity of stateful computations: the circled stateful computation of version 2 is more involved, and cannot be executed atomically by the available stateful ALU, while the stateful computations of version 1 can be. This is because the ALU considered here (the Read/Write

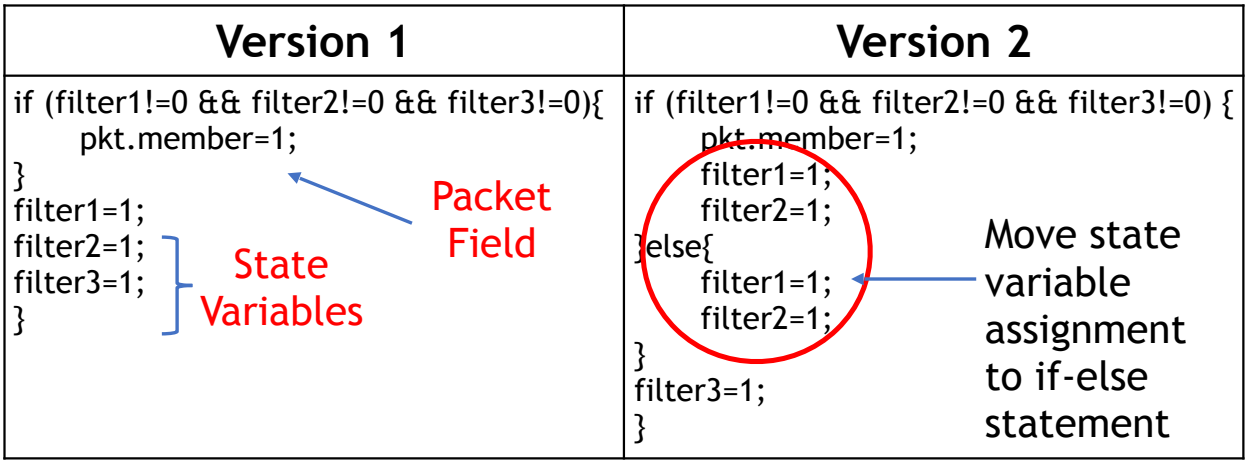

Figure 3.2: Two semantically equivalent versions of a Domino program [130]. Version 1 compiles; version 2 fails to.

ALU [130]) can only handle the atomic update of one state variable, but not the atomic update of two variables as required by version 2. Essentially, the compiler is running into a computational limit.

This difference in DAGs for 2 semantically equivalent programs occurs because Domino's compiler passes are program rewrite rules that repeatedly transform the program in an attempt to find a simpler version of it (i.e., the DAG representation) that readily maps to switch ALUs. However, these rewrite rules are *incomplete*: they do not find a semantically-equivalent version of the DAG that *can* map all nodes to the available ALU type (i.e., version 1's DAG) when given the version 2 program. In effect, the compiler's rules do not fully explore the search space of machine code programs that could implement the high-level program.

Although the specific situation in Figure 3.3 could be fixed by a compiler developer through the addition of another rewrite rule, similar situations will continue to emerge in the future. In fact, rule-based compilers and programs can be thought of as two sides of an arms race, with compilers getting more complex over time to incorporate more rewrite rules that simplify more complex program patterns. By contrast, as we next discuss, by exhaustively searching the space of machine code, program synthesis has the potential to provide a simpler and more future-proof compiler design.

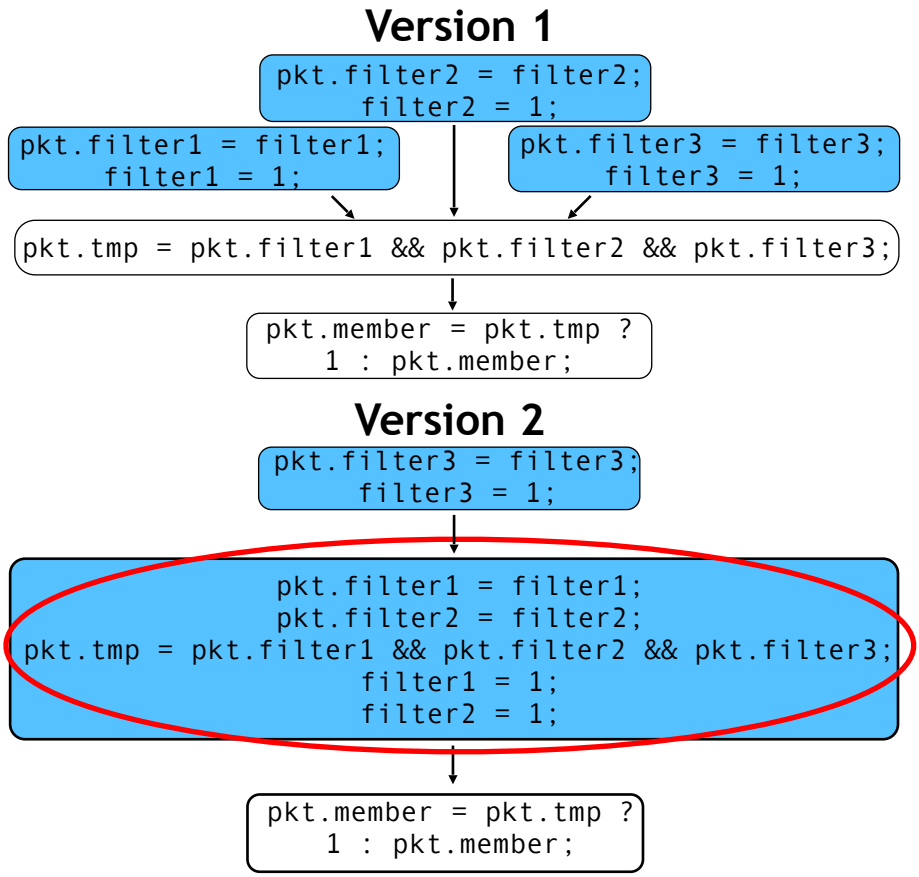

Figure 3.3: Simplified Domino DAGs for both versions. Stateless computations are unshaded; stateful are shaded. The circled node shows the large amount of atomic stateful computation in version 2.

The case for program synthesis. To address the incompleteness of rule-based compilation, we first observe that the search for machine code for a given high-level program can be thought of as a combinatorial search problem called *program synthesis*. In program synthesis, we are looking for a program *implementation* that is semantically-equivalent to a program *specification:* on all legal inputs, the outputs of the specification and implementation agree. In our context, implementations are drawn from the set of all machine code programs that can be implemented on a 2D ALU grid of bounded size. The specification is the high-level language program that must be compiled. The benefit of the program synthesis approach is that synthesis engines can search a family of implementations using efficient algorithms [70, 134]. If at least some implementations meet the specification, the synthesis engine is far more likely to find it than a rule-based compiler.

Hence, synthesis greatly reduces butterfly effects similar to Figure 3.2.

However, without domain-specific techniques to prune the search space of implementations, synthesis can quickly become intractable in practice. Thus our goal is to leverage synthesis for compilation because of its ability to search the machine code space, while keeping compilation times reasonable using domain-specific techniques to prune the search space. In this paper, we aim for compilation times of an hour, although most of our programs compile in a few minutes (§3.7.1). While an hour seems excessive, we believe that it is better than the alternative of having developers tweak programs manually, which requires low-level hardware expertise, is error prone, and may take even longer. Further, there must be some price to pay for higher quality code; other compiler techniques (e.g., link-time optimization [88]) exhibit a similar tradeoff.

3.4 Code generation as synthesis

Chipmunk takes as inputs: (1) a packet transaction and (2) a specification of the pipeline's capabilities. It produces machine code for that pipeline, which consumes a small number of resources (ALUs and pipeline stages). We first describe how we produce machine code given a fixed pipeline depth (stages) and width (ALUs per stage) (§3.4.1–§3.4.5). We then show how to use this to find code with small depth and width (§3.4.6).

3.4.1 Code Generation Using SKETCH

We briefly describe SKETCH, the program synthesis engine we use in this paper. SKETCH takes two inputs: a specification and a *sketch*, a partial program with *holes* representing unknown values in a finite range of integers. Sketches constrain the synthesis search space by only considering for synthesis those programs in which each sketch hole is filled with an integer belonging to the hole's range. Sketches encode human insight into the shape of synthesized programs. SKETCH then fills in all holes with integers so that the completed sketch meets the specification,

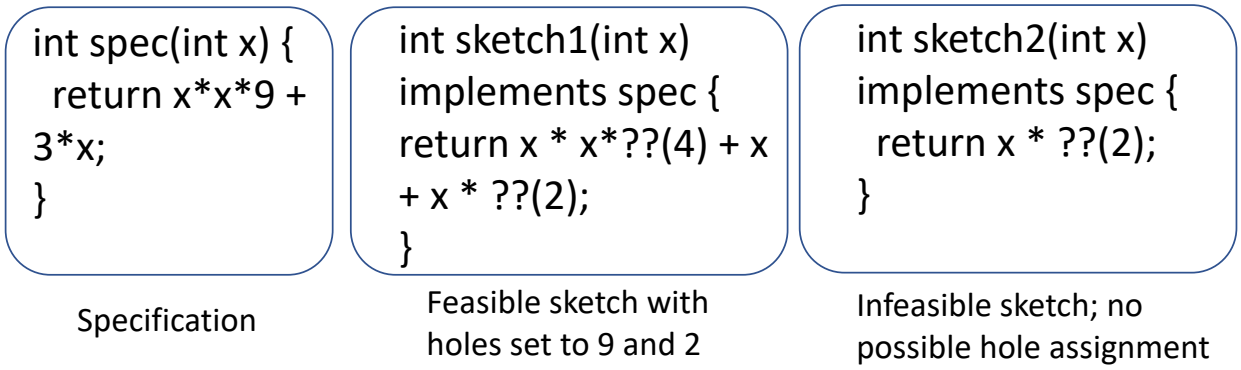

Figure 3.4: Synthesis in SKETCH. ??(b) is a hole with a value in $[0, 2^b - 1]$

assuming it is possible to meet the specification (Figure 3.4), or says that it is impossible to do so. § 1.3.2 describes SKETCH's internals. To build a code generator using SKETCH, we need to determine an appropriate set of holes, the sketch, and the specification.

Holes. The code generator needs to ultimately choose the right value of low-level programmable hardware knobs (Table 3.1), e.g., which inputs to wire to each ALU (the input muxes), what operation each performs (ALU opcodes), which PHV container is used for the outputs (output muxes), which packet field is allocated to each container, etc. Our goal is to get SKETCH to make these choices. Hence, we encode these programmable knobs as SKETCH holes.

Sketch. We use a sketch to represent the space of valid pipeline configurations to be considered for implementing packet-processing programs. This sketch captures all the possible computations supported by the pipeline as a function of the holes. First, the sketch models the effect of each stateful and stateless ALU on PHV containers and ALU-local state, as a function of holes corresponding to the ALU opcode, input mux controls, immediate operands, and local state variable(s). Second, the sketch models the pipeline: the flow of PHVs from stage to stage, as a function of holes corresponding to the input/output mux controls and field to PHV allocations. In effect, our pipeline sketch is a programmatic representation of the 2D grid of ALUs in Figure 3.1; see § 1.3.1 for an example.

Programmable knob / SKETCH hole	Hole bit width
ALU opcode (e.g., +,-, *, /)	$log_2(\|opcodes\|)$
Input mux control: which PHV feeds an ALU	$log_2(\|PHVs\|)$
Output mux control: which ALU feeds a PHV	$log_2(\|ALUs\|)$
Indicator bit to track if a field is allocated to a PHV container	$\|fields\| * \|PHV\|$
Indicator bit to track if a state variable is allocated to a stateful ALU	$\|state\_vars\| * \|num\_stages\| * \|alus\_per\_stage\|$
Immediates/constants in instructions	constant bit width

Table 3.1: Programmable knobs and their hole bit width

3.4.2 Packet Transactions as Specifications

We now show how we encode specifications. We use the terms packet (state) vector to refer to a vector, every dimension of which corresponds to a single packet field (state variable) from the specification, i.e., the packet transaction in Figure 3.1. We use the term state+packet vector to represent the concatenation of the state and packet vectors. Our goal is to synthesize a pipeline implementation that respects the packet transaction specification on an arbitrary input *packet trace*: on any sequence of packet vectors and arbitrary initial state vector, the output sequence of packet vectors and final state vector must be the same for the specification and the implementation. We call this problem *trace-based synthesis*.

However, trace-based synthesis appears daunting since packet traces can be infinitely long, while SKETCH is designed to work with finite inputs. But we can reduce trace-based synthesis to a simpler finite problem we call *packet-based synthesis,* where our goal is to synthesize a pipeline implementation such that on any arbitrary *single* packet vector and arbitrary *single* previous state vector, the updated state+packet vector after processing that packet must be the same for the specification and the implementation.

To perform this reduction, we need to establish that a solution to packet-based synthesis is also a solution to trace-based synthesis, which follows by induction on the length of the packet

sequence. We note that packet-based synthesis is a stronger requirement than trace-based synthesis because it requires agreement on any possible previous state vector, even if such a state vector might never occur in a state sequence starting with an arbitrary initial state vector. This reduction is also similar to a classic technique used in hardware verification [57], where a sequential equivalence-checking problem of matching a sequence of outputs between a specification and an implementation is reduced to a combinational equivalence-checking problem, which requires both the outputs and the states to match at every time step. This reduction is practically significant because synthesis tools can deal with the finite input space of a single state vector and a single packet vector.

We convert the packet-processing program written as a packet transaction in Domino into a SKETCH specification that takes as input a state+packet vector and outputs an updated state+packet vector. To convert the Domino program into a SKETCH specification, we added a pass to the Domino compiler [42]. This is relatively straightforward because both Domino and SKETCH have a very similar C-like syntax. Replacing Domino with P4-16 @atomic as the input language for Chipmunk would similarly need a p4c [100] compiler pass.

3.4.3 The Slicing Technique

While packet-based synthesis is simpler than trace-based synthesis, it is still too slow on several benchmarks (Table 3.3). To speed up synthesis further, we developed a technique called *slicing*. We start with a simplified version of slicing that only works for stateless and deterministic packet transactions. We then refine it to handle state and randomness.

To motivate slicing, observe that in packet-based synthesis, we require the specification and implementation to agree on the *entire* updated state+packet vector. Instead of requiring agreement on the entire vector, we factorize the requirement into a collection of simpler requirements or *slices*. Each slice is a simpler synthesis problem that synthesizes a pipelined implementation such that the implementation and the specification agree only on a single vector dimension of the

updated state+packet vector (e.g., only `pkt.sample` or only `count` in Figure 3.1). Once we have successfully synthesized each slice, we merge together the slice implementations by stacking the resulting pipeline implementations on top of each other to form the final hardware implementation.

Slicing has two main advantages over packet-based synthesis. First, each slice can be synthesized in parallel because each slice implementation runs on an independent sub-grid of the pipeline with no overlap between the sub-grids. Second, because each slice's implementation only satisfies a subset of the specification, i.e., one dimension instead of all dimensions of the state+packet vector, it can be synthesized using a sub-grid with smaller size than would be needed for the original specification. A smaller grid requires fewer holes to be synthesized (Table 3.1), reducing the synthesis time (Table 3.3) for any one slice relative to the original specification.

Handling state modifications. When the packet transaction modifies state, the above slicing algorithm is no longer correct. To see why, consider the specification "`count++; pkt.f = count;`". This specification sets a packet field, `pkt.f`, to the most recent value of a switch counter, `count`, which increments on every packet. This problem will be factorized into two slices: one each for `pkt.f` and `count`. In the first slice, we require an implementation that sets `pkt.f` to the previous `count + 1`. In the second slice, we require an implementation that sets `count` to the previous `count + 1`.

Note, however, the first slice *does not* require `count` itself to be updated to its correct final value. This means that the first slice can produce implementations that simply set `pkt.f` to the previous `count + 1`, without actually ever updating `count`! The result is that when a trace of multiple packets has passed through the pipeline, `pkt.f` will always be set to 1, i.e., the initial value of `count` (0) + 1. This is because `count` is never updated in the first slice. However, a correct implementation should set `pkt.f` to 1, 2, 3, . . . , over successive packets.

More generally, for any slice S, suppose there is a state variable i such that S's packet field or state variable depends on i (e.g., `pkt.f` depends on `count`). Then we say i influences S. Then

as part of the slice S, we should also require that i be updated in the implementation to match up with how the specification updates i. Otherwise, i may not be updated and S's packet field or state variable cannot make use of the updated i for its own computation.

Hence, for each slice, we additionally assert that the specification and implementation also agree on any state variables that influence that slice's packet field or state variable. §3.4.4 proves that this additional assert produces the correct behavior of all slices in the presence of state—as long as state modifications complete within a clock cycle as described in §3.2. In our example above, this additional assert ensures that `count` is also set to previous `count + 1` in the first slice, in addition to `pkt.f` being set to previous `count + 1`.

Non-determinism. The use of randomness (e.g., hashing) within a packet transaction can result in the merged implementation (after slicing) differing in behavior from the packet transaction. This is because when a random-number-generating computation is duplicated in two or more slices, we cannot guarantee that the random numbers generated in each slice will be identical. This can be fixed by either seeding the random number generators in all slices to the same value from the control plane or precomputing such random numbers and storing them in packet fields before executing the packet transaction. We follow the second approach in this paper.

The cost of slicing. Slicing does not exploit opportunities to share computations between different slices. For instance, in our example, the update to `count` is duplicated across the two slices. Thus slicing requires additional ALUs and PHV containers for these redundant computations. In our evaluations, we find that programs do not use too many containers/ALUs in the first place (< 10) and using slicing adds at most 3 containers and ALUs per stage (Table 3.3). For context, RMT has about ~200 ALUs/PHV containers per stage [30]. This is a reasonable trade-off for faster compilation.

3.4.4 Correctness of slicing

We now prove the correctness of the slicing technique. We first introduce some notation before proving correctness.

Definition 3.1. Spec denotes the packet transaction specification for a single packet. It is a function with inputs comprising a packet vector $\vec{p}$ and a state vector $\vec{s}$; and with outputs comprising an *updated* packet vector and an *updated* state vector.

Individual fields in the output vector of the function can be accessed by member name or member index. For example, $Spec(\vec{p}, \vec{s}).m$ denotes the member m of the output of the function **Spec** on inputs $\vec{p}$ and $\vec{s}$ and $Spec(\vec{p}, \vec{s})[i]$ denotes the i^{th} member in the output vector of the function.

Definition 3.2. Spec* denotes the packet transaction specification for a sequence of packets. It is a function with inputs comprising a sequence of n packets $\{\vec{p_n}\}$, and an initial state vector $\vec{s_0}$; and with outputs comprising a final packet vector and a final state vector after the n^{th} packet is processed by the **Spec** function. It is defined inductively as follows:

$$\text{Spec}^*(\{\vec{p}_1\}, \vec{s_0}) = \text{Spec}(\vec{p}_1, \vec{s_0})$$
$$\text{Spec}^*(\{\vec{p}_n\}, \vec{s_0}) = \text{Spec}(\vec{p}_n, \text{Spec}^*(\{\vec{p}_{n-1}\}, \vec{s_0}).\vec{s})$$

where $\{\vec{p}_i\}$ denotes the first i packets of the input packet sequence.

Note that in the inductive step, Spec* applies the Spec function to the n^{th} packet in the sequence and the output state resulting from applying Spec* inductively.

Definition 3.3. Impl is a function with inputs comprising a packet vector $\vec{p}$ and a state vector $\vec{s}$, and with outputs comprising an *updated* packet vector and an *updated* state vector, as reflected by the functionality of the programmable switch.

Definition 3.4. Impl* is a function with inputs comprising a sequence of n packets $\{\vec{p_n}\}$ and an initial state vector $\vec{s_0}$, and with outputs comprising a final packet vector and a final state vector after the n^{th} packet is processed by the programmable switch. It is defined inductively as follows:

$$\text{Impl}^*(\{\vec{p}_1\}, \vec{s_0}) = \text{Impl}(\vec{p}_1, \vec{s_0})$$
$$\text{Impl}^*(\{\vec{p}_n\}, \vec{s_0}) = \text{Impl}(\vec{p}_n, \text{Impl}^*(\{\vec{p}_{n-1}\}, \vec{s_0}).\vec{s})$$

Definition 3.5. Influence: Consider the set $S = \{\vec{p}, \vec{s}\}$. We say that $u \in S$ influences $v \in S$, if there exist c_1 and c_2 such that $Spec(\{\vec{p}, \vec{s}\}/\{u\}, u = c_1).v \neq Spec(\{\vec{p}, \vec{s}\}/\{u\}, u = c_2).v$ (or) if there exists $inter \in S$ such that u influences $inter$ and $inter$ influences v.

Let **I(i)** denote a vector of *indices* of state variables that influence the output of the i^{th} packet field, and let **NI(i)** denote a vector of *indices* of state variables that do *not* influence the output of the i^{th} packet field.

Definition 3.6. Slicing-based synthesis

$$\forall i \, \forall \vec{p} \, \forall \vec{s}, \;\; \text{Spec}(\vec{p}, \vec{s})[i, I(i)] = \text{Impl}_i(\vec{p}, \vec{s})[i, I(i)]$$

where i represents the i^{th} packet field and $I(i)$ is as defined above. We refer to Impl_i as a slicing-based implementation function for the i^{th} packet field. The final Impl needs to *concatenate* all slicing-based implementation functions to ensure the correctness of all packet fields' output. As an example with 2 packet fields, Impl consists of both Impl_0 and Impl_1. Therefore, we can say that Impl_i are "stacked" on top of each other to form Impl.

Definition 3.7. Trace-based synthesis for each slice

$$\forall i \, \forall \{\vec{p}_n\} \, \forall \vec{s_0}, \;\; \text{Spec}^*(\{\vec{p}_n\}, \vec{s_0})[i, I(i)] = \text{Impl}_i^*(\{\vec{p}_n\}, \vec{s_0})[i, I(i)]$$

Slicing-based synthesis only ensures the correct implementation for a particular slice on ONE input packet. However, here we want the semantic equivalence when taking any packet trace, which is a sequence of packets, as an input to the program.

Theorem 3.8. ***Slicing-based synthesis*** $\implies$ ***Trace-based synthesis for each slice.***

Proof. We will prove this for any i (i.e., for all i^{th} packet fields), and by induction on n, the number of packets in the sequence of packets processed by the switch.

For the base step, $n = 1$, we have:

$$\text{Spec}^*(\{\vec{p}_1\}, \vec{s}_0)[i, I(i)] = \text{Spec}(\vec{p}_1, \vec{s}_0)[i, I(i)] \quad \ldots (\text{def of Spec}^*)$$

$$\text{Impl}_i^*(\{\vec{p}_1\}, \vec{s}_0)[i, I(i)] = \text{Impl}_i(\vec{p}_1, \vec{s}_0)[i, I(i)] \quad \ldots (\text{def of Impl}_i^*)$$

By the definition of slicing-based synthesis,

$$\text{Spec}^*(\{\vec{p}_1\}, \vec{s}_0)[i, I(i)] = \text{Impl}_i^*(\{\vec{p}_1\}, \vec{s}_0)[i, I(i)]$$

Induction hypothesis: Assume that the claim holds for $n = k$:

$$\forall i \, \forall \{\vec{p}_k\} \forall \vec{s}_0, \quad \text{Spec}^*(\{\vec{p}_k\}, \vec{s}_0)[i, I(i)] = \text{Impl}_i^*(\{\vec{p}_k\}, \vec{s}_0)[i, I(i)] \tag{3.1}$$

Induction step: Now we prove the claim for $n = k + 1$.

$$
\begin{aligned}
&\mathrm{Spec}^*(\{\vec{p}_{k+1}\}, \vec{s}_0)[i, I(i)] \\
&= \mathrm{Spec}(\vec{p}_{k+1}, \mathrm{Spec}^*(\{\vec{p}_k\}, \vec{s}_0).\vec{s})[i, I(i)] \quad \ldots \text{(def of Spec}^*\text{)} \\
&= \mathrm{Impl}_i(\vec{p}_{k+1}, \mathrm{Spec}^*(\{\vec{p}_k\}, \vec{s}_0).\vec{s})[i, I(i)] \\
&\quad \ldots \text{(def of } \textbf{Slicing-based synthesis}\text{)} \\
&= \mathrm{Impl}_i(\vec{p}_{k+1}, \mathrm{Spec}^*(\{\vec{p}_k\}, \vec{s}_0).\vec{s}[I(i)], \mathrm{Spec}^*(\{\vec{p}_k\}, \vec{s}_0).\vec{s}[NI(i)])[i, I(i)] \\
&\quad \ldots \text{(split state variables into influence and non-influence parts)} \\
&= \mathrm{Impl}_i(\vec{p}_{k+1}, \mathrm{Impl}_i^*(\{\vec{p}_k\}, \vec{s}_0).\vec{s}[I(i)], \mathrm{Spec}^*(\{\vec{p}_k\}, \vec{s}_0).\vec{s}[NI(i)])[i, I(i)] \\
&\quad \ldots \text{(by equation (3.1))} \\
&= \mathrm{Impl}_i(\vec{p}_{k+1}, \mathrm{Impl}_i^*(\{\vec{p}_k\}, \vec{s}_0).\vec{s}[I(i)], \mathrm{Impl}_i^*(\{\vec{p}_k\}, \vec{s}_0).\vec{s}[NI(i)])[i, I(i)] \\
&\quad \ldots \text{(the variables in NI(i) cannot influence the variables in I(i) or i,} \\
&\quad \text{so we can set them to any value without affecting the final output)} \\
&= \mathrm{Impl}_i(\vec{p}_{k+1}, \mathrm{Impl}_i^*(\{\vec{p}_k\}, \vec{s}_0).\vec{s})[i, I(i)] \\
&\quad \ldots \text{(merge state variables with indices in I(i) and NI(i))} \\
&= \mathrm{Impl}_i^*(\{\vec{p}_{k+1}\}, \vec{s}_0)[i, I(i)] \quad \ldots \text{(def of Impl}_i^*\text{)}
\end{aligned}
$$

□

Finally, if we stack Impl_i on top of each other to form Impl, this Impl can ensure the semantic equivalence for any input packet trace.

3.4.5 Other Optimizations

Scaling up synthesis to larger input ranges. SKETCH is designed to synthesize implementations that meet the specification on a small range of inputs for each input variable (e.g., all x

values between 0 and 31 in Figure 3.4). Scaling SKETCH to synthesize implementations that meet the specification on larger ranges of inputs (e.g., all 32-bit integers) is challenging. This is because the SAT solver within SKETCH isn't optimized for rapid verification on large input ranges for two reasons. First, SKETCH's unary encoding is particularly inefficient in its use of memory [1, 132]. Second, a SAT solver does not contain many of the theories available in a full-blown SMT solver such as Z3 [146]. Z3 is better suited for verifying spec-implementation equivalence over large input ranges because it contains specialized decision procedures for integers and bit vectors that scale to larger input ranges.

To address this problem, we decouple the input ranges for synthesis and verification, adapting an idea proposed in prior work [76]. We use SKETCH to synthesize a solution for a small input bit width of 2, i.e., all packet fields and state variables are assumed to take values between 0 and $2^2 - 1$. Then, we take the resulting completed sketch (i.e., with all holes filled in with integers) and use Z3 to verify it on a larger input bit width (currently all 10-bit integers). If Z3 finds a counterexample, we rerun SKETCH again, using asserts to rule out the previously obtained hole values (hole elimination) or using the newly found counterexample to create an additional concrete input on which the specification and the sketch must agree (counterexample assertion). Between hole elimination and counterexample assertion, we find that counterexample assertion performs much better because it can rule out not only the hole assignment that led to the current Z3 verification counterexample, but also all other hole assignments that could have potentially led to that counterexample.

Constant synthesis. One challenge for program synthesis tools is synthesizing holes with large bit ranges. In our context, these are immediate operands for ALUs, which can be up to 32 bits wide. The difficulty of synthesizing such large holes has been documented before [3, 118]. With SKETCH, we also observed a steep increase in synthesis time when using holes with bit widths exceeding 15 bits [1]. To synthesize large holes for immediate operands, we developed an algorithm based on using a dynamic pool of constants from which SKETCH picks immediate

operands. Our algorithm initializes this pool to all constants that appear in the input packet transaction. In addition, we augment the pool with all numbers in a small range of integers (0–3). If Z3 verification fails, we *update* the pool with packet field and state variable values appearing in the counterexample produced by Z3. This algorithm is very efficient but incomplete—the main source of incompleteness in Chipmunk—because it only samples a few integer values; hence, it can fail to find a large hole when one actually exists. However, empirically, we find that it performs well because it adapts to the supplied program and learns from counterexamples.

Canonicalization. SKETCH needs to allocate packet fields to PHV containers and state variables to stateful ALUs, while respecting the hardware constraint that no PHV container or stateful ALU is oversubscribed. There are many feasible allocations that satisfy this constraint. However, in a symmetric grid, where the same type of ALU is tiled out over the entire grid and each ALU can use any PHV container as an operand, many of these allocations are equivalent to each other. We can use this symmetry to speed up synthesis.

In a symmetric grid, for PHV allocation, we rename packet fields so that they have canonical names $f_1, f_2, \ldots$ following [21]. Then, we allocate f_1 to container 1, f_2 to container 2, and so on. This allocation is as good as any other because in a symmetric grid all containers are equivalent in their abilities (i.e., an ALU can use any of these containers as an operand and can output to any of these containers). In other words, any allocation can be canonicalized by renaming variables.

For state variables, the situation is similar, but with one important difference. It doesn't matter which stateful ALU within a stage a state variable is allocated to due to symmetry. However, it *does* matter which stage's ALU a state variable is allocated to. This is because of dependencies within the feed-forward pipeline: an update to a state variable in a later stage can depend on the value of a state variable in an earlier stage, but the reverse is disallowed in a feed-forward pipeline. Thus state allocation exhibits symmetry within ALUs in one stage, but not across ALUs of different stages. Hence, we still use SKETCH to determine which stage a state variable should go into, but assign the state variable to a canonical stateful ALU within that stage.

Hole elimination vs. counterexample assertion. We considered different modes in which Z3 and SKETCH can cooperate in code generation. Our intuition was that counterexample assertion would lead to faster compile time because hole elimination only eliminates the particular hole assignment that caused that Z3 failure, while a new counterexample would eliminate *all* the hole assignments that could have caused that failure. Our experiments (Table 3.5) confirm this intuition. We set a timeout for the hole elimination to 10× the time for counterexample assertion to limit run time of our experiments; without a timeout, the benefits of counterexamples would be even more pronounced. Hence, we use counterexample assertion.

Canonical vs. synthesized allocation. We study two ways in which state variables can be allocated to stateful ALUs and packet fields can be allocated to PHV containers. In canonical allocation, as discussed in §3.4.5, a packet field is always allocated to its canonical container and a state variable is always allocated to its canonical ALU after SKETCH determines its stage. In synthesized allocation, on the other hand, we disregard symmetry, and ask SKETCH to find the allocation from scratch. Hence, in synthesized allocation, SKETCH determines which container to use for a field, and which stage and which stateful ALU in that stage to use for a state variable. Table 3.6 shows the results. We find that canonical allocation and synthesized allocation perform similarly on benchmarks that have a short compilation time, but that canonical allocation can significantly reduce time on the longer benchmarks.

3.4.6 Reducing Grid Size by Parallel Search

So far, we have focused on code generation for an ALU grid of a certain depth and width. To reduce resource usage, we need to find a small grid to implement each slice of the packet transaction. To do so, we independently solve a synthesis problem for each combination of slice and grid size in parallel (Figure 3.5). For a packet transaction to be successfully synthesized, all its slices must be successfully synthesized at some grid size (which may be different for each slice). Thus, if a transaction can be successfully synthesized, the time to successfully synthesize that

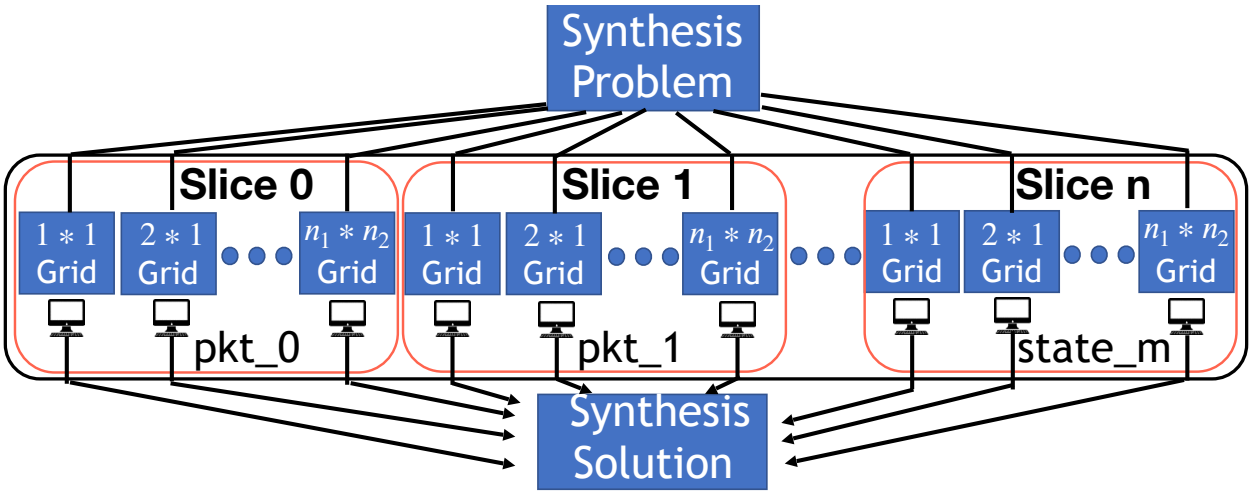

Figure 3.5: Parallel search to reduce grid size

transaction is the maximum of the times to successfully synthesize each of its slices. For each slice, the time to successfully synthesize that slice is the minimum of the times to successfully synthesize that slice across all grid sizes searched. We set an upper bound on the grid size and a timeout on the synthesis time for any one of the parallel synthesis problems. For each parallel synthesis problem, we internally use SKETCH's parallel mode [70]; thus, there are two levels of parallelism. Our parallel search strategy over grid sizes does not guarantee the smallest possible grid size for a problem, but builds on our observation that smaller grid sizes generally lead to faster synthesis times, if the slice can actually fit into the smaller grid. We have not currently implemented a full system to run synthesis problems in parallel and emulate it using sequential execution in our evaluations. However, we believe it will be straightforward to implement such a system given the embarrassingly parallel nature of our search.

3.5 Retargetable code generation

The techniques in the last section can generate hole-value assignments for a packet-processing program written in a high-level language, given a sketch of the pipeline. Two problems remain: (1) a sketch must be developed for the pipeline; and (2) the hole-value assignments must be

$l \in$ literals $\quad v \in$ variables	bin_op	$\in$ binary ops $\quad un\_op \in$ unary ops
$t \in$ ALU type declaration	::=	`stateful` | `stateless`
$d \in$ packet field declarations	::=	list of variables v
$h \in$ hole declarations	::=	list of variables v
$sv \in$ state variable declarations	::=	list of variables v
$e \in$ expressions	::=	$l \mid v \mid e\ bin\_op\ e \mid un\_op\ e$
		| `Mux`$(e, \ldots)$
$s \in$ statements	::=	$e = e \mid s\ ;\ s \mid$ `return` (e)
		| `if` (e) $\{s\}$ | `if` (e) $\{s\}$ `else` $\{s\}$
$p \in$ alu specification	::=	$t; d; h; sv; s$

Figure 3.6: ALU DSL. Mux is an input/output mux.

mapped to a format that the hardware/backend understands.

Unfortunately, we found that developing a sketch of the pipeline manually is error-prone for several reasons (§3.5.1). Hence, we developed a *pipeline description language*, a declarative specification of a pipeline ALU's compute capabilities and the interconnection between these ALUs (§3.5.1). We designed a *pipeline sketch generator* (§3.5.2) that takes specifications in this language and automatically produces a sketch of the pipeline. Thus, this pipeline description language enables *retargetable code generation* [39, 85]: Chipmunk can generate code for a number of distinct packet-processing pipelines, given a description of each pipeline.

Pipeline descriptions were also directly useful in targeting the two backends that we support, a simulator for the Banzai machine model [2] and the Tofino ASIC [114]. In particular, we were able to use declarative pipeline specifications to automatically generate executable Banzai *behavioral models* (§3.5.3), which were helpful in debugging Chipmunk itself. We also leveraged the pipeline specification language to produce a Tofino-specific code generator that "lifts" the low-level hole-value assignments from Chipmunk to a surface language (i.e., P4-14) accepted by Tofino's compiler (§3.5.4).

3.5.1 Pipeline Description Language

Writing a pipeline sketch manually is hard for three reasons.

1. **Large sketches:** Even for modest grid sizes (e.g., a 3-by-3 pipelined grid), the total number of hole bits and the number of lines in the sketch file often exceeds a few hundred.
2. **Different ALU capabilities on different targets:** There are significant differences in the capabilities of ALUs in the different pipelines we were experimenting with: a simulator of a switch pipeline (Banzai [130]), the Tofino switch [114], and subsets of the ALU functionalities provided by either. We found that constructing a pipeline sketch manually for each of these three different cases—and each ALU within each case—was highly error-prone.
3. **Diverse grid interconnects among ALUs:** The specific connectivity among ALUs and PHVs differs from one pipeline to another. For example, Banzai provides all-to-all connectivity between all PHVs and ALUs: an ALU operand can come from any PHV. However, for wiring efficiency, some switching chips only provide all-to-all connectivity within clusters of PHVs and ALUs: an ALU operand can only come from a PHV in the same cluster as that ALU.

For these reasons, we developed two domain-specfic languages (DSLs) to write switch pipeline specifications, one each for (1) *computation* (e.g., opcodes of an ALU) and (2) *communication* (e.g., the interconnection between these ALUs to create a pipeline through input and output muxes). Thus, the computation DSL specifies *local* intra-ALU features of a switch pipeline, while the communication DSL specifies *global* inter-ALU features of a switch pipeline. We expect these specifications to be written once at switch design time by the backend compiler developer, not repeatedly by the programmer who writes Domino programs (Figure 3.7).

DSL to describe ALU computation. We developed an ALU DSL (Figure 5.6) to specify the computational capabilities of a single switch ALU, i.e., the programmable knobs available in the ALU (Table 3.1). Figure 5.6 shows the grammar of the ALU DSL. The DSL allows a developer to specify

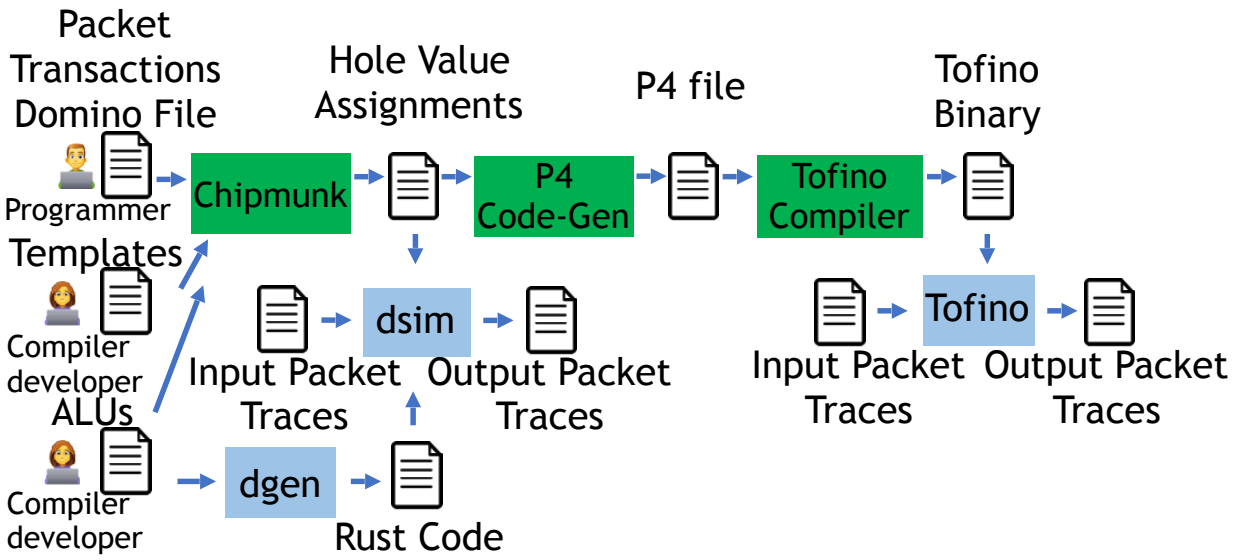

Figure 3.7: Workflow of Chipmunk

the number of ALU operands, where each operand comes from (i.e., packet field, state, immediate operand), the ALU's operation over its operands in the form of simplified C-like code, and what value(s) the ALU must return. Our DSL is expressive in its ability to express diverse target ALUs. It can express all the stateful and stateless instructions proposed as Banzai atoms [130], the stateless and stateful ALUs in the Tofino ASIC [114], as well as subsets of the functionality of each of these ALUs.

DSL to describe ALU interconnect. We specify the interconnections between ALUs in a grid using a *grid template* written in the Jinja2 template language [74]. A grid template is a skeleton of a sketch for a pipeline grid, representing the scaffolding or glue required to connect together ALUs, such as wires and muxes connecting PHVs to ALU inputs, and ALU outputs to PHVs (Figure 3.1). The grid template has placeholders to hold SKETCH code for ALUs (generated from the ALU DSL) and can repeat ALUs for a given width and depth.

3.5.2 Pipeline Sketch Generation

Chipmunk's pipeline sketch generator takes an ALU DSL program and a grid template as input and generates a sketch corresponding to the ALU, repeating ALU code as necessary to fill out the 2D grid specified in the grid template. Chipmunk then feeds the generated sketch to SKETCH, which returns a value for each hole or says that the sketch is infeasible. If it is infeasible, we return a compile error. If it is feasible, we use the hole-value assignments to program the backends (§3.5.3, §3.5.4, and §3.6).

3.5.3 Producing Behavioral Models

A declarative pipeline description language enables the *automatic generation of pipeline behavioral models* for the Banzai machine model from pipeline DSL specifications, akin to P4-bmv2 [145]. We designed a behavioral-model-generator, `dgen` [152], which takes as input a switch pipeline specification in our DSLs (§3.5.1) and generates Rust code that simulates the action of that pipeline on packets. The Rust code when built produces an executable version of the pipeline `dsim` [152], which can consume and output packets, manipulating both the packets and internal pipeline state, serving as a behavioral model for the pipeline. Thus, `dsim` allows us to observe the input-output behavior of machine code (e.g., ALU opcodes, mux settings, etc.) produced by Chipmunk.

We used `dsim` to fuzz-test Chipmunk using random test packets for 100+ packet transactions drawn from our programs (§3.7), i.e., test whether Chipmunk generates correct pipelined machine code from packet transactions. We did this in three steps. First, we created random packet vectors as test inputs. Second, we directly executed the packet transaction on these test packets to record its input-output behavior without pipelining. We performed direct execution by writing each packet transaction as a Rust function and running the Rust function repeatedly on the test packets. Third, we compare the behavior from direct execution with the behavior `dsim` produces

on the hole-value assignments generated by Chipmunk for the same packet transaction. If the two behaviors are different, it points to a bug in Chipmunk's code generation. So far, our fuzz testing has not yet revealed any bugs.

3.5.4 Lifting to Switch Surface Languages

Leveraging a DSL for expressing ALUs also enabled us to translate the outputs from Chipmunk's synthesis into an input language supported by the Tofino switch compiler [102], P4-14. We did this in two steps. First, we developed a P4-14 template program in Jinja2 [74], with placeholders for P4 registers and tables, which have a one-to-one correspondence with the 2D grid of ALUs from the pipeline sketch. For instance, each stateless ALU corresponds to a P4-14 primitive action, each stateful ALU corresponds to a P4 extern [101], and each state variable corresponds to a P4-14 register.

Second, we filled in the placeholders in the P4-14 template by translating the low-level integer-valued hole-value assignments outputted by synthesis into higher level P4-14 code accepted by the Tofino compiler. In particular, we translated holes from stateless ALUs (opcodes, operands) and stateful ALUs (opcodes, operands, and choice of output PHVs) into P4-14 code by leveraging a traversal of the abstract syntax tree (AST) of the ALU expressed in our DSL. For instance, let's say the ALU DSL file contains a function that takes three parameters: two operands (A and B) and an opcode. The hole-value assignments provide the value of the opcode, say 3, which stands for the + operation. Then, we can traverse the AST corresponding to the function and simplify the function to A+B, by treating the opcode as a constant (3) and applying constant propagation on the AST.

3.6 Experiences with Tofino

After filling in the placeholders, the P4 program is given to the Tofino compiler to generate a Tofino binary. However, this can sometimes result in an incorrect implementation due to a subtle interplay between P4's sequential semantics and the Tofino hardware's parallel operation. To see why, consider a "swap" packet transaction that swaps the value of two packet fields (top and bottom) without using any temporary fields. This transaction can be implemented in Tofino using a single pipeline stage with 2 ALUs (top and bottom). The top stateless ALU transfers the bottom PHV to the top PHV and the bottom stateless ALU does the reverse, using an add opcode in each ALU with 0 as one operand. Chipmunk can also generate hole-value assignments in our behavioral model for this one-stage implementation of the swap transaction.

Now, how do we realize this swap in P4? We can create two P4 tables, one each for the top and bottom ALU, and use P4-14 `apply` statements to execute each table's ALU on incoming packets. However, the `apply` statement has sequential semantics. Because each table reads a field (top/bottom) that the other writes (bottom/top), sequential semantics will force the Tofino compiler to infer a read-after-write dependency between the two tables. Hence, the Tofino compiler will place the tables in two consecutive stages, not one. This is not just wasteful in stages, it is incorrect relative to what we want: it results in both top and bottom taking the same value instead of swapping values. Here, the Tofino compiler is correctly respecting P4-14's sequential semantics for `apply` statements, but the behavior is different from what Chipmunk expects from parallel execution of the ALUs. In other words, it is hard to express the intra-stage parallelism required for operations like swap directly in P4, despite the hardware supporting it.[1]

To address the gap in the abstraction levels between Chipmunk's needs and P4-14, we use a compiler pragma to instruct the Tofino P4 compiler to ignore all table dependencies that it

[1]We note that Tofino does support a swap primitive that can be directly invoked by the programmer as an intrinsic function without writing a swap program in P4. However, the broader point illustrated by our programmer-written swap still holds: code generation requires us to express intra-stage parallelism, which is challenging.

finds on its own. We instead enforce all dependencies ourselves. Chipmunk already handles dependencies because dependencies need to be respected to generate machine code that agrees with the specification. To enforce dependencies that Chipmunk finds, we use a second compiler pragma. This pragma instructs the Tofino compiler to place a table containing a stateful/stateless ALU in the same stage that the ALU belonged to in Chipmunk's output.

Reflections. Considerable research has looked at raising the level of abstraction of languages for networking. On the other hand, when building the Tofino backend, we needed to *lower* the level of abstraction to enforce low-level control over the hardware using pragmas. Pragmas are effectively a mechanism to get the Tofino compiler out of the way—to perform fewer program analyses and make fewer modifications. We could have avoided pragmas and created a simpler backend if Tofino supported direct assembly programming. We hope our results make a case for switching chip vendors to support such low-level interfaces to their chips.

3.7 Evaluation

Our evaluation answers the questions listed below.

- **How does synthesis-based compilation compare to rule-based compilation?** (§3.7.1) We investigate this using the Chipmunk and Domino compilers for the Banzai target, on the metrics of (i) ability to compile programs successfully, (ii) resource usage (i.e., pipeline stages and ALUs per stage) of successfully compiled programs, and (iii) compile time.
- **Can synthesis effectively target a switching ASIC? (§3.7.2)** We show experimental results using Chipmunk to target Tofino.
- **How beneficial are slicing and the other optimizations in synthesis-based compilation? (§3.7.3)**

Choice of baseline. For our baseline compiler, we used the Domino rule-based compiler because it can also take as input a high-level specification in transactional style, similar to Chipmunk. Domino also has a few classical compiler optimizations baked into it (e.g., common subexpression elimination, strength reduction, fusing code segments, etc.) [42]. We note that Domino also uses SKETCH for the final step of code generation after much preprocessing using classical rewrite rules [130, §4.3]. As our results show, using SKETCH so late in code generation doesn't help significantly with compiler quality. By contrast, Chipmunk treats the entire code generation problem as a program synthesis problem, with little preprocessing.

Comparing with a commercial compiler like the Tofino compiler [102] would have been preferable to comparing with a research prototype like Domino. However, we can not directly compare with the Tofino compiler because it does not support a transactional style of programming in either of its frontend languages (P4-14 and P4-16). Instead, with the Tofino compiler, the programmer has to manually partition code into different tables and then chain together the tables to implement their desired feature—in other words, program at a lower level than the P4-16 `@atomic` or packet transactional style. We note, however, that we have observed the same butterfly effects that motivated our work (§3.2) with the Tofino compiler as well. Our results make a case for including program synthesis within commercial compilers, allowing them to support a similar transactional programming style.

Benchmarks. We collected 14 benchmarks (Table 3.2) from multiple sources [77, 98, 130]. These were previously written as Domino programs using the packet transactions abstraction [130]. All of these benchmarks are known to successfully compile with Domino using one of the 7 stateful atoms combined with the stateless atom proposed by Banzai [130]. We model each of these 7+1 atoms as an ALU using our ALU DSL. We verified that these benchmarks can indeed be successfully compiled with Chipmunk using the same ALU that was used for Domino.

We further create semantic-preserving rewrites of the benchmarks, which we call *mutations*, e.g., Figure 3.2. Comparing Chipmunk and Domino using mutations allows us to measure

whether the two compilers can still compile mutations of an original program that was itself successfully compiled. The mutations can also be used to compare Domino and Chipmunk's resource consumption on a larger set of programs than we started out with.

To create mutations, we added a compiler pass to the Domino compiler to modify Domino programs in semantic-preserving ways. This pass repeatedly transforms programs by randomly picking one of three transformations. The pass modifies (1) `if(x) B else A` into `if(!x) A else B` , (2) `if(A and B)` into `if (B and A)`, and (3) `if(x)` into `if(x and 1==1)`. These mutations are simplistic and are not fully representative of the diverse ways in which developers craft programs that are semantically equivalent. Yet, even with these mutations, we demonstrate (§3.7.1) that Domino fails to compile many mutations, while Chipmunk successfully compiles all of them.

Experimental setup. We used a single stateless ALU type for all experiments regardless of which stateful ALU we used. This stateless ALU is modeled after the stateless atom proposed in Banzai [130]. For the stateful ALU, we used the same stateful Banzai atom for each benchmark as reported in the benchmark's source [77, 98, 130]. Unless stated otherwise, we run Chipmunk with slicing (§3.4.3) and all other optimizations (§3.4.5) enabled. We use SKETCH's parallel mode, which takes advantage of multi-core parallelism [70].

Recall that Chipmunk uses parallel search over different slices and grid sizes (§3.4.6) by running different synthesis problems on different machines. Additionally, each machine needs multiple cores for SKETCH's parallel mode. We do not have a cluster of physical machines readily available and the cost of running unoptimized Chipmunk on EC2 is prohibitive. Instead, we used a single 32-core 64-hyperthread 256-GB RAM machine (AMD Opteron 6272) to run Chipmunk with the Banzai ALUs (§3.7.1 and §3.7.3) and used a single 28-core 56-hyperthread 64-GB RAM machine (Intel Xeon Gold 6132) to run Chipmunk with the Tofino ALUs (§3.7.2), and report compilation times emulating the parallel search strategy from §3.4.6. That is, the compile times we present in §3.7.1, §3.7.2, and §3.7.3 were obtained by sequential experiments, with the compile

Program	Chipmunk compile rate	Domino compile rate	Chipmunk depth, width	Domino depth, width	Chipmunk compile time (s)	Domino compile time (s)	Banzai ALU [130]
BLUE (increase) [45]	100%	0%	4,6	N/A	213	N/A	**pred raw**
BLUE (decrease) [45]	100%	0%	4,6	N/A	1134	N/A	**sub**
CONGA [10]	100%	0%	1,7	N/A	16	N/A	**pair**
Flowlet switching [128]	100%	100%	3,8	8.3,4	280	1.5	**pred raw**
Learn filter [130]	100%	100%	3,8	17.5,4	291	2.1	**raw**
Marple new flow [98]	100%	0%	2,3	N/A	12	N/A	**pred raw**
Marple TCP NMO [98]	100%	0%	3,5	N/A	15	N/A	**pred raw**
RCP [142]	100%	100%	2,9	5.6,5	34	2	**pred raw**
Sampling [130]	100%	0%	2,2	N/A	33	N/A	**if else**
SNAP heavy hitter [19]	100%	100%	1,3	3.3,3	70	1.2	**pair**
Spam Detection [19]	100%	80%	1,3	3.1,3	51	7	**pair**
Stateful firewall [19]	100%	100%	4,8	15.5,4.1	7020	1	**pred raw**
STFQ [52]	100%	0%	2,7	N/A	36	N/A	**nested if**
DNS TTL change [26]	100%	0%	3,10	N/A	223	N/A	**nested if**

Table 3.2: Compile rate, time, and resources averaged over 10 mutations; BLUE means better compilation rate; ALU names refer to Banzai's atoms.

time for a given ALU grid size being the maximum across all per-slice compile times, and the per-slice compile time being the minimum compile time across all ALU grid sizes for that slice. When we don't use slicing, the compile time is simply the minimum compile time across all ALU grid slices. We also estimate the monetary cost of running Chipmunk with slicing on Amazon EC2.

3.7.1 Synthesis *vs.* Rule-Based Compilation

For each of the 14 benchmarks, we created 10 mutations using our mutating compiler pass. This gives us 140 programs to compare Domino and Chipmunk on, using the following metrics: (i) what % of the mutations of an original program can the compiler successfully compile? (we call this the *compile rate*), (ii) how many pipeline stages and ALUs per stage are needed to fit the program?, (iii) how long does it take for successful compilation? We average across 10 mutations for each benchmark.

We report our results in Table 3.2. We find that (i) Chipmunk can compile all the mutations we produced, while Domino fails in many cases, (ii) Chipmunk's average compilation times are longer than Domino's, but largely fit into our time budget of ~1 hour (§3.3), and (iii) when both

Program	Compile time (s)			Depth, width	
	Slicing	Orig.	speedup	Slicing	Orig.
BLUE (increase) [45]	213	2792	13.11×	4,6	4,4
BLUE (decrease) [45]	1134	32400	28.57×	4,6	4,4
CONGA [10]	16	16	1×	1,7	1,6
Flowlet switching [128]	280	61035	217.98×	3,8	4,7
Learn filter [130]	291	291	1×	3,8	5,6
Marple new flow [98]	12	12	1×	2,3	2,3
Marple TCP NMO [98]	15	16	1.07×	3,5	3,4
RCP [142]	34	96	2.82×	2,9	3,6
Sampling [130]	33	33	1×	2,2	2,2
SNAP heavy hitter [19]	70	75	1.07×	1,3	1,2
Spam Detection [19]	51	62	1.22×	1,3	1,2
Stateful firewall [19]	7020	>86400	>12.31×	4,8	N/A
STFQ [52]	36	1795	49.86×	2,7	4,9
DNS TTL change [26]	223	>86400	>387.44×	3,10	N/A

Table 3.3: Resource usage, compile time with and without slicing, averaged over 10 mutations. 1 day timeout.

Chipmunk and Domino can both compile a mutation, Chipmunk requires fewer pipeline stages and slightly higher ALUs per stage than Domino. However, stages are far more constrained resources (e.g., 32) compared to ALUs in each stage (e.g., 224) [30].

The quality benefits of synthesis also come with a monetary cost: Chipmunk requires more compute resources than Domino (§3.4.6). However, this cost is reasonable. We use some typical numbers to estimate the cost of a Chipmunk compilation when implementing the strategy from §3.4.6 in the cloud. To estimate the degree of parallelism with slicing, we use some typical numbers from our benchmarks. Assuming 5 slices per program across both packet fields and state variables, and 10 grid sizes to be searched for each slice (i.e., up to a 3*3 grid, which is the largest grid size we search), we require around 50 VMs. We pick the m5.16xlarge spot EC2 instance [13] with 64 vCPUs and 256 GB RAM because it is closest to our local machine. With per-second billing, a one-minute minimum billing time [17], and a typical synthesis time of 5 minutes (Table 3.2), the compilation cost is roughly $2.66, with the fairly pessimal assumption that all 50 VMs

Program	Depth, width	Chipmunk compile time (s)
BLUE(increase) [45]	2, 5	112
BLUE(decrease) [45]	2, 6	113
CONGA [10]	1, 7	6
Flowlet switching [128]	2, 7	95
Marple new flow [98]	1, 2	5
Marple TCP NMO [98]	2, 4	8
RCP [142]	1, 8	26
Sampling [130]	1, 2	24
SNAP heavy hitter [19]	1, 3	40
DNS TTL change [26]	2, 8	34

Table 3.4: Compiling original benchmarks to Tofino.

are occupied the entire 5 minutes. We note that the alternative of rule-based compilers will cost much more in hourly developer wages [82] due to compilation failures.

3.7.2 Compilation to Tofino Switching ASIC

After modeling the Tofino stateful and stateless ALUs using the ALU DSL (§3.5.1), we were able to compile and run 10 out of our 14 original benchmarks (without mutations) on Tofino. We report the resource consumption and the compilation times in Table 3.4. Compilation times are well within an hour for all benchmarks. The times in Tables 3.4 and 3.2 are different because (1) the ALUs are different (Banzai vs. Tofino) and (2) different machines were used.

However, we were unable to compile 4 benchmarks to Tofino. The mutations in Table 3.2 were theoretically guaranteed to compile to *some* Banzai atom because the original programs compiled to that atom; we do not have any such guarantees with Tofino as these benchmarks have not been compiled to Tofino before. In fact, for all 4 of these benchmarks, the resulting sketch file for at least one slice was infeasible up to a grid size of 2*3. Looking closer, we noticed that these 4 benchmarks need a more complicated condition for conditional state updates than supported by Tofino—a computational limit (§3.3). Hence, we suspect these benchmarks may not be able to compile to *any* grid size given our Tofino ALU model. However, we cannot yet prove

that these programs cannot be compiled to even an infinite grid of a certain ALU type. Proving such "unrealizability" is an area of research in synthesis [62].

3.7.3 Benefits of Optimizations

3.7.3.1 No Slicing vs. Slicing

We now compare Chipmunk's performance with and without slicing on two metrics: pipeline resource usage and compilation time. We keep all other optimizations enabled. Table 3.3 shows the benefits of slicing for the Banzai backend. We observe that slicing provides a few orders of magnitude speedup on several benchmarks; this is primarily because slicing a program allows us to fit the program within a smaller grid, which translates into a smaller search space/time for SKETCH. Slicing causes a small increase in ALUs per stage because slicing does not share computations between slices. However, Chipmunk with slicing still consumes fewer stages than Domino (Table 3.2). Beyond performance, slicing also helped us debug the generated P4 programs on Tofino. Each slice could be tested independently on a small grid—instead of testing the whole program on a larger grid. This enabled us to localize and fix bugs in P4 code generation faster.

Even with slicing, some benchmarks (BLUE (decrease) and Stateful firewall) still incur a long synthesis time. To speed these up, we can involve the programmer in synthesis and have them set some holes in the generated sketch (e.g., ALU opcodes, output muxes) based on their insight into the program. For BLUE (decrease) and Stateful firewall, we observed speedups of 4.18 × and 37.14 × relative to the Chipmunk times in Table 3.2 by intelligently setting the value of only 3% of all the holes, hinting at the promise of an interactive approach to further reduce compile time.

3.7.3.2 Hole elimination vs. counterexample assertion

We considered different modes in which Z3 and SKETCH can cooperate in code generation. Our intuition was that counterexample assertion would lead to faster compile time because hole

Program	Cex assertion (s)	Hole elimination (s)
BLUE (increase) [45]	213	>2130
BLUE (decrease) [45]	1134	>11340
CONGA [10]	16	16
Flowlet switching [128]	280	>2800
Learn filter [130]	291	291
Marple new flow [98]	12	12
MARPLE TCP NMO [98]	15	15
RCP [142]	34	>340
SAMPLING [130]	33	>330
SNAP heavy hitter [19]	70	>700
Spam Detection [19]	51	>510
Stateful firewall [19]	7020	>70200
DNS TTL change [26]	223	438
STFQ [52]	36	36

Table 3.5: Compilation time for Hole elimination vs. counterexample assertion in SKETCH-Z3 loop. BLUE means faster compilation speed.

Program	Canonicalized (s)	Synthesized (s)
BLUE (increase) [45]	213	273
BLUE (decrease) [45]	1134	1056
CONGA [10]	16	18
Flowlet switching [128]	280	1112
Learn filter [130]	291	355
Marple new flow [98]	12	19
MARPLE TCP NMO [98]	15	23
RCP [142]	34	46
SAMPLING [130]	33	59
SNAP heavy hitter [19]	70	70
Spam Detection [19]	51	51
Stateful firewall [19]	7020	9810
DNS TTL change [26]	223	616
STFQ [52]	36	4803

Table 3.6: Compilation time in seconds for synthesized vs. canonical allocation. BLUE means faster compilation speed.

elimination only eliminates the particular hole assignment that caused that Z3 failure, while a new counterexample would eliminate *all* the hole assignments that could have caused that failure. Our

experiments (Table 3.5) confirm this intuition. We set a timeout for the hole elimination to 10× the time for counterexample assertion to limit run time of our experiments; without a timeout, the benefits of counterexamples would be even more pronounced. Hence, we use counterexample assertion.

3.7.3.3 Canonical vs. synthesized allocation

We study two ways in which state variables can be allocated to stateful ALUs and packet fields can be allocated to PHV containers. In canonical allocation, as discussed in §3.4.5, a packet field is always allocated to its canonical container and a state variable is always allocated to its canonical ALU after SKETCH determines its stage. In synthesized allocation, on the other hand, we disregard symmetry, and ask SKETCH to find the allocation from scratch. Hence, in synthesized allocation, SKETCH determines which container to use for a field, and which stage and which stateful ALU in that stage to use for a state variable. Table 3.6 shows the results. We find that canonical allocation and synthesized allocation perform similarly on benchmarks that have a short compilation time, but that canonical allocation can significantly reduce time on the longer benchmarks.

Overall, with all other optimizations and slicing enabled, we observed an average 6.21× speedup with counterexample assertion vs. hole elimination and an average 11.02× speedup with canonicalization.

3.8 Limitations and Future Work

We now briefly discuss Chipmunk's main limitations along with avenues for future work. First, while Chipmunk rejects far fewer programs than Domino (Table 3.2), it is still incomplete and can reject feasible programs. In particular, because slicing has an additional cost PHV/ALU cost, a sliced packet transaction may no longer fit into a small grid, while the original packet

transaction may have. Additionally, our constant synthesis algorithm is also incomplete. Second, even after slicing, Chipmunk's compile times are still much longer than Domino. We believe there is still room to improve Chipmunk by exploiting dependencies between parts of the packet transaction, similar to Domino's use of computation DAGs (Figure 3.3). As shown in §3.7.3, interactively involving the programmer can also substantially speed up synthesis-based compilation; this is another area that we plan to explore. Third, Chipmunk uses significant compute resources for its parallel grid search (§3.4.6) to reduce compile time; reducing this resource usage by packing synthesis runs into fewer machines is another area for future work. Fourth, currently all our benchmarks are relatively small programs. For future work, we hope to scale Chipmunk sufficiently to perform code generation for much larger production-quality programs such as switch.p4 [106]. Fifth, Chipmunk is currently restricted to code generation and does not concern itself with the problem of allocating memory for large stateful arrays or match-action tables. Combining ILP [75] or SMT [46] techniques for memory allocation with synthesis for code generation is another area for future work.

3.9 Related Work

Synthesis has been applied to synthesize network updates [98, 120], routing table configurations from high-level policies [58, 138], policies from configurations [27], and control planes [139]. These efforts target network configurations and policies pertaining to access control, reachability, and isolation. In contrast, Chipmunk uses program synthesis to generate packet-processing code for programmable switches. Program slicing [151] computes a subset of program statements that can influence a variable's value at some program location. It has been applied to debugging [151], network verification [108, 113], and query optimization [11]. Chipmunk applies slicing in the new context of machine code generation for switches, requiring considerable adaptation of the basic slicing idea (§3.4.3).

3.10 Summary

We presented Chipmunk, a program-synthesis-based compiler for switches. Chipmunk fits programs into limited switch pipeline resources—programs which might otherwise be rejected by rule-based compilers. To do so, it leverages domain-specific synthesis techniques to expedite compilation and uses a pipeline description language to target multiple backends. We hope that the techniques and results we have presented will stimulate follow-on research in designing program synthesis algorithms that compile programs faster and with fewer compute resources, and produce machine code with lower switch resource consumption. We also believe the ideas here are more generally applicable to FPGA [65, 92] and ASIC-based [64] SmartNIC pipelines.

4 | CaT: A Solver-Aided Compiler for Packet-Processing Pipelines

Compiling high-level programs to high-speed packet-processing pipelines is a challenging combinatorial optimization problem. The compiler must configure the pipeline's resources to match the semantics of the program's high-level specification, while packing all of the program's computation into the pipeline's limited resources. State of the art approaches tackle individual aspects of this problem. Yet, they miss opportunities to produce globally high-quality outcomes within reasonable compilation times.

We develop a framework to decompose the compilation problem for such pipelines into three phases—making extensive use of solver engines (e.g., ILP, SMT, and program synthesis) to simplify the development of these phases. *Transformation* rewrites programs to use more abundant pipeline resources, avoiding scarce ones. *Synthesis* breaks complex transactional code into configurations of pipelined compute units. *Allocation* maps the program's compute and memory to the pipeline's hardware resources.

We prototype these ideas in a compiler, CaT[1], which targets (1) the Tofino programmable switch pipeline and (2) Menshen, a cycle-accurate simulator of a Verilog description of the RMT pipeline. CaT can handle programs that existing compilers cannot currently run on pipelines and generates code faster than existing compilers, where the generated code uses fewer pipeline

[1]CaT stands for **C**ode Generation **a**nd **T**able Allocation.

resources.

4.1 Introduction

Reconfigurable packet-processing pipelines (e.g., RMT [30]) are emerging as important programmable platforms. They are embodied in many programmable high-speed switches and network interface cards (NICs) such as the Tofino [68], Trident [33], and Jericho switches [32]; the Pensando SmartNIC [14]; and Intel IPUs [69].

Programmable pipelines are organized into multiple stages, where each stage processes one packet in parallel, and hands it off to the next stage (§4.2.1). Each stage contains memory blocks to hold tables containing packet-matching rules and state (e.g., counters) maintained across packets. Header fields are extracted from packets to match the table rules. Once the packet's fields are matched against a rule, the packet or state can also be updated using an action.

P4 [104] is emerging as a popular language to program these pipelines. P4 offers the ability to parse packets according to custom header definitions, and specify the match types and actions on parsed packets. A P4 action may modify packet headers and switch state.

The compilation problem. The networking community has developed P4 programs targeting programmable pipelines for several research [23, 71, 91, 95] and production [61, 78, 87, 107] use cases. To enable these use cases, a compiler must translate P4 programs to pipeline configurations. This compiler must solve a combinatorial optimization problem with several challenging aspects to it:

(1) Multiple resource types: There are multiple pipeline resources, with some resources being scarce, e.g., pipeline stages, and others being abundant, e.g., arithmetic logic units (ALUs). Some resources must be allocated hand in hand (e.g., match memory and ALUs).

(2) Transactional guarantees: P4 actions can be annotated to have transactional guarantees [4, 105]: executing to completion on each packet before processing the next one. If such a transac-

tional P4 action requires multiple pipeline stages, the compiler must be able to split the action into multiple ALUs and stages, ensuring the implementation respects the action's transactional semantics [130].

(3) All-or-nothing fit: A program for a high-speed pipeline can either run at the pipeline's highest throughput (typically line rate above a minimum packet size), or cannot run at all. Thus, it is important to "pack" all of the P4 program into the pipeline's limited resources.

Prior work has tackled several individual aspects of this compilation problem (§4.2.2). Such an approach loses opportunities to globally reduce resource usage (e.g., stages), which is necessary to fit complex programs on a pipeline. However, it is challenging to solve a single combinatorial optimization problem. Our goal is to find a good decomposition of the large problem into smaller pieces, enabling global optimization of resource usage, while keeping each piece small enough to solve efficiently.

Our approach. In this paper, we present an end-to-end compiler, CaT, that unifies prior approaches and translates high-level P4 programs into a low-level representation suitable for pipelined execution. We take inspiration from *high-level synthesis* (HLS)—a technology for improving productivity of hardware design for ASICs [141] and FPGAs [67].

Informally, HLS [37] takes as input a high-level algorithmic description of the hardware design with no reference to clocks or pipelining, and with limited parallelism in the description. An HLS compiler then progressively lowers this high-level description down to an optimized hardware implementation, pipelining the implementation if possible, executing multiple computations in parallel, scheduling computations in time, and converting these computations into a register-transfer level (RTL) design.

We believe such an approach to developing compilers targeting packet-processing pipelines will raise the user's level of programming abstraction, while retaining the performance of low-level pipeline programming. For a user developing algorithmic programs in P4 (such as those used for in-network computation, e.g., [71, 122, 155]), such an approach eliminates the labor of

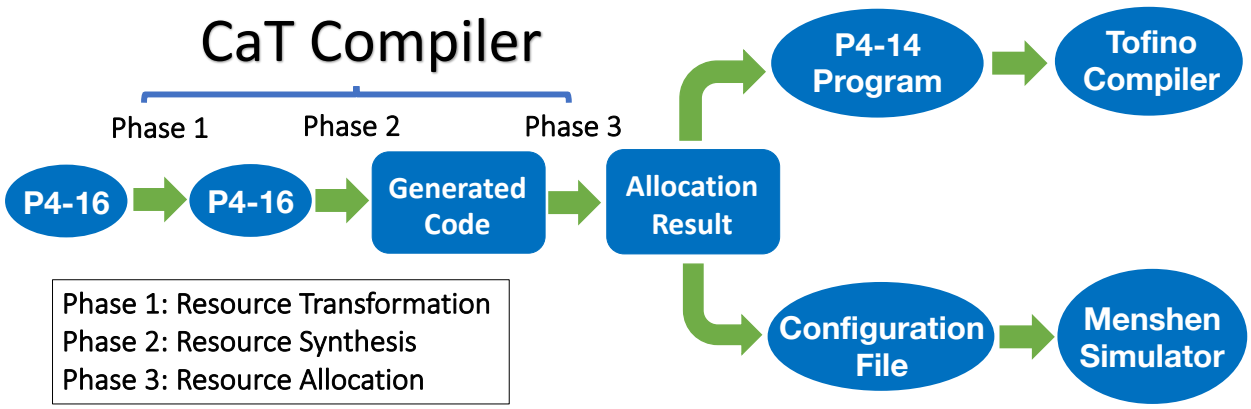

Figure 4.1: The workflow of the CaT compiler.

manually breaking the high-level algorithmic computation into actions spread over many pipeline stages (§4.3).

The workflow of our compiler, CaT, is shown in Figure 4.1. It consists of three phases. The input consists of P4 code containing tables that match on specific headers and action code blocks that modify packet headers and state. The action code blocks may be written without regard to their feasibility within a single pipeline stage. The first phase of CaT employs *resource transformations* that rewrite a high-level P4 program to another semantically-equivalent high-level P4 program; these rewrites are used to transform a computation's use of one scarce resource to its use of a relatively abundant resource, and potentially reduce the number of stages as well. The second phase performs *resource synthesis* to lower transactional blocks of statements in the high-level P4 program to a lower-level program suitable for hardware execution. In this step, individual ALUs in hardware are configured to realize the programmers' intent in the transactional action blocks, while respecting the ALUs' computational limits. The third phase performs *resource allocation* to allocate the computation units corresponding to the lowered program to physical resources such as ALUs and memory in the pipeline. Notably, our compiler workflow works within the confines of the widely used P4 ecosystem without requiring the development of a new domain-specific language (DSL) for packet processing.

Our contributions. The main technical contribution of CaT's three-phase approach is the modu-

larization of the large combinatorial optimization problem of compilation into smaller problems, whose solutions still enable a high-quality global result (§4.7). These smaller problems can also be fed to solver engines, simplifying the process of solving them. Additionally, we improve upon the state of the art and introduce new techniques in each phase. In particular, our resource transformations (§4.4) are driven by a novel *guarded dependency analysis* that identifies false dependencies between computations, exposing more parallelism opportunities when rewriting programs to use more abundant resources. Our resource synthesis phase (§4.5) uses a novel synthesis procedure that quickly finds pipelined solutions with good-quality results for complex actions. It separates out stateful updates from stateless updates, to decompose a large program synthesis problem into smaller and more tractable subproblems; each subproblem uses a program synthesis engine (SKETCH) as a subroutine. Stateless code is synthesized into a minimum-depth computation tree, i.e., with the minimum number of stages. In comparison to prior work [50], this new synthesis algorithm allows CaT to handle many large actions, in a much shorter time, and with fewer computational resources needed for compilation. Finally, our resource allocation phase (§4.6) uses a constraint-based formulation that extends prior work [75] to handle complex multi-stage transactional actions; this formulation can be fed to either an ILP or SMT solver. Our techniques can support general P4 programs (including @atomic constructs [105]) efficiently, including programs translated into P4 from higher-level DSLs developed for pipeline programming [46, 50, 60, 130, 135].

Our prototype of CaT can target: (1) the Tofino pipeline, and (2) an open-source RMT pipeline called Menshen (that was previously implemented on an FPGA) [94, 150]. Existing commercial switches have proprietary instruction sets that preclude the kind of low-level resource allocation and control over ALU configurations needed by CaT. Therefore, our backend for Tofino [68] generates low-level P4 in lieu of machine code. To evaluate CaT in full generality, we extend Menshen's open-source register-transfer level (RTL) Verilog model with additional resources for our experiments. We generate code for the cycle-accurate simulator of Menshen, and also use

it for testing the CaT prototype. Our results (§4.7) show that CaT can automatically compile programs that previously required manual changes to be accepted by the Tofino compiler. On other challenging benchmarks, CaT produces good quality code and does so about 3 times faster (on average) than prior work [50].

4.2 Background and Related Work

4.2.1 Packet-Processing Pipelines

The compiler target in this paper is a programmable packet-processing pipeline following the Reconfigurable Match Tables (RMT) architecture [30]. Such pipelines are present in commercially available programmable switches such as the Barefoot Tofino, Broadcom Trident, and Mellanox Spectrum, and NICs such as the Pensando DPU. An RMT-style pipeline consists of (i) a programmable packet parser and (ii) a number of processing stages structured around match-action computation. We describe these components below.

A programmable parser takes in a programmer-specified header specification, and extracts packet header fields. This set of fields is termed the *packet header vector (PHV)*. PHV fields can be both read and written in each pipeline stage, termed a match-action stage. One match-action stage extracts relevant fields from the PHVs using a crossbar circuit. The fields are then matched against user-inserted rules in stage-local match memory. The memory may also contain *state*, i.e., values maintained on the switch and updated by every packet, such as a packet counter. Once a packet matches a rule, a corresponding set of actions is invoked. The actions are implemented using Very Long Instruction Word (VLIW) ALUs which may modify multiple PHV fields in one shot. Some match-action tables may be skipped entirely (e.g., due to control flow) through hardware components called *gateways*.

Three factors limit the available resources and expressiveness of packet-processing pipelines.

First, to support high throughput (e.g., 6.5 Tbit/s in Tofino), pipelines are clocked at high frequencies (e.g., 1 GHz for Tofino). Thus, the pipeline must admit a new packet every clock cycle. Hence, stateful computations (read-modify-write) must finish in one clock cycle. Second, on-chip and stage-local memories are limited in size, to support fast lookup. Third, constraints on chip area and power limit the number of pipeline stages (e.g., 12 match-action stages in Tofino) and control circuitry (e.g., number of gateways and crossbars). Such exacting hardware constraints pose compiler challenges. Furthermore, program behavior is *all-or-nothing*: a program that fits into the pipeline resources would run at the pipeline's clock frequency; otherwise it cannot be run. There is no graceful degradation between these extremes.

4.2.2 Related Work

There has been significant interest in developing compilers and domain-specific languages (DSLs) for packet-processing pipelines. We categorize the existing compiler efforts based on their support for program rewriting, code generation, and resource allocation.

DSLs for programming packet pipelines P4 and NPL are the most widely used languages to program packet pipelines. They share many syntactic and semantic aspects. Several academic projects have proposed new DSLs or extensions to P4 to remedy many of P4's shortcomings. For instance, microP4 [137] adds modularity to P4. Lyra [46] addresses the issue of portability of programs across multiple devices. Lyra and FlightPlan [140] address the problem of partitioning a program automatically across multiple devices. Lucid [135] introduces an event-driven programming model for control applications in the data plane. P4All [60] extends P4 to support 'elastic' data structures, whose size can grow and shrink dynamically based on the availability of switch resources. Domino [130] is a DSL that supports transactional packet processing: a programmer specifies a block of code that is executed on each packet in isolation from other packets. These languages can all be translated into P4, and in this paper, we directly take P4 programs as our

starting point. Thus, our work is complementary to work on such new DSLs. One limitation of CaT is that it does not currently handle the problem of partitioning a *network-wide* program into per-device programs, like Lyra and Flightplan. Instead, our goal is to build a high quality compiler that inputs a P4 program for a *single device* and outputs a high-quality implementation for that device.

Program rewriting The open-source reference P4 compiler [100], which is the foundation for most P4 compilers including the widely-used Tofino compiler [68], employs rewrite rules to turn an input P4 program into successively simpler P4 programs. These rewrite rules consist of classical optimizations like common sub-expression elimination and constant folding. Rewrite rules are also employed by Cetus [86] and Lyra [46] to merge tables in different stages (under certain conditions) into a single "cartesian-product" table in a single stage, thereby saving on the number of stages. CaT uses rewrite rules to transform uses of scarce resources (gateways, stages) to more abundant ones (tables, memory, ALUs), in a style similar to Cetus.

Code generation for complex actions Domino [130] and Chipmunk [50] tackle the problem of *code generation*: selecting the right instructions (i.e., ALU opcodes) for a program action expressed in a high-level language. These compilers have to respect the limited capabilities of each stage's VLIW ALUs while correctly implementing state updates according to @atomic semantics for transactions (§4.2.1). Domino largely uses rewrite rules and employs program synthesis to code-generate just the stateful fragments in the action, but minor semantic-preserving modifications to programs can cause compilation to fail. Chipmunk addresses this drawback of Domino by using program synthesis to exhaustively search for ALU configurations that could implement a high-level program, but at the expense of high compile time. Lyra [46] uses *predicate blocks,* chunks of code predicated by the same path condition, to break up algorithmic code into smaller blocks that have only inter-block (but no intra-block) dependencies. CaT's resource synthesis is faster than Chipmunk's and more reliable than Domino's (Table 4.5, §4.7.3). It generalizes Lyra's predicate block approach by considering ALUs expressed via a parameterizable grammar, such

Project	Program Rewriting	Code Generation	Resource Allocation	Retargetability		New Language Constructs
				Instruction Sets	Resource Constraints	
Domino [130]	Yes	Rewriting, some program synthesis		Atom templates		Packet transactions
Chipmunk [50]		Program synthesis		ALU DSL		Packet transactions
Lyra [46]	Yes		SMT		SMT constraints	Network-wide programs
Flightplan [140]		Resource rules	Resource rules		Resource rules	Network-wide programs
Cetus [86]	Yes		Table Merging, PHV Sharing		SMT constraints	
P4All [60]			ILP		ILP constraints	Elastic data structures
Jose et al. [75]			ILP		ILP constraints	
Lucid [135]		Memops		Memops		Event-driven programming
Tofino compiler [68]	Yes	Yes	Heuristics			
CaT (this work)	Yes	Min-depth tree synthesis	ILP/SMT	ALU grammars	ILP/SMT constraints	P4's atomic construct

Table 4.1: CaT unifies prior work in P4 compilers (first column) to provide and improve various features (listed in other columns) in an end-to-end flow, and does so within the context of the P4 language without needing a new DSL.

that the procedure is independent of the operations in the program's source code or intermediate representation.

Resource allocation The problem of allocating specific resources required by a P4 program (e.g., match memory blocks, a specific number of ALUs, etc.) can be posed as an integer linear programming problem (ILP) [60, 75] or as a constraint problem [46] for Satisfiability Modulo Theory (SMT) solvers [22]. If the constraints of the hardware are modeled precisely, ILP-based techniques can improve resource allocation relative to greedy heuristics for resource allocation. To this end, CaT's resource allocation (§4.6) uses a fine-grained constraint-based formulation that models detailed pipeline resources and enables global optimization by considering dependencies across tables as well as within actions.

CaT compiler phase	CaT technique	Builds on prior work	Differences in CaT	Other complementary work
1: Resource transformation	Rewrite rules	LLVM [84], HLS [37, 67], p4c [100]	Rewrite rules target RMT, based on novel guarded dependency analysis	p4c [100] uses platform-independent rewrites, Cetus [86] merges tables
2: Resource synthesis	Mapping operations to ALU pipeline	HLS operation binding [37, 67]	Stateful updates restricted to 1 stage	Lucid [135] uses syntactic rules to ensure operations map to Tofino
	Synthesis procedure uses SKETCH queries for program synthesis	Chipmunk [50]	Novel synthesis procedure: faster, more scalable, uses smaller queries	
	Target portability: via parameterizable grammars for ALUs	Sketch [132], Chipmunk [50]	Generate resource graph (used in Phase 3), not low-level ALU configs	
	Preprocessing: branch removal, SSA, SCC in computation graph	Domino [130], SSA [38], VLIW [83]	No backward control flow (similar to Domino)	
	Simplifications: const prop, expr simplification, deadcode elimination	LLVM [84]	No backward control flow	
3: Resource allocation	Constraints for match memories	Jose et al. [75], Lyra [46]	Associates match memories with corresponding action resources (ALUs)	
	Constraints for multi-stage actions	HLS scheduling [37], Domino [130], Chipmunk [50]	Uses result of Phase 2 for intra-action dependencies and ALU output propagation	
	Constraints for multiple transactions	Domino [130], Chipmunk [50] handle a single transaction only	Enforces inter-table and intra-action dependencies for global optimization	
	Modeling real hardware constraints in backend FPGA target	Menshen provides a FPGA backend target [150]	Extended functionality of resources in comparison to Menshen	Tofino compiler uses heuristics

Table 4.2: Detailed relationship of the 3 phases of CaT with prior work on compilers, HLS, and packet-processing pipelines.

4.3 CaT: Motivation and overview

Motivation. Today, P4 developers typically write down actions in P4 programs with the assumption that each action must finish in one stage. However, tracking the hardware-level feasibility of an action leads to thinking at an unnecessarily low level of abstraction, especially when developing high-speed algorithmic code. Consider the example pseudocode (motivating example ME-1) shown in Figure 4.2. This function implements the SipHash algorithm, used as a hash function to prevent collision-based flooding attacks [20]. The developer of a P4 version of this algorithm (distinct from the authors of this paper) started with a high-level transactional description of the algorithm (Table 3, [155]). The developer then *manually* changed it into a pipelined implementation (Table 4, [155]), because the algorithm as expressed cannot be compiled by the Tofino compiler since it cannot be finished in one stage. We believe that a good compiler should automate this process of *synthesizing* pipelined implementations from transactional specifications. Indeed, CaT can successfully handle this example (discussed in §4.7.3), without requiring an expert developer to manually pipeline their code. Furthermore, beyond automatically pipelining a single transaction, a compiler should be capable of pipelining *multiple* such transactions, generating pipeline configurations for their resulting implementations, and then *allocate* physical resources in the pipeline for these implementations. Finally, P4 programs can often be written in different ways, which consume different kinds of resources; if possible, a compiler must be able to transform uses of a scarce resource into uses of an abundant resource, e.g., using larger tables in lieu of more stages [86].

CaT's approach. CaT is an end-to-end compiler for P4-16 programs that takes inspiration from high-level synthesis (HLS) [67, 141] to provide both: (1) a high level of abstraction for specifying packet-processing functionality, and (2) high quality of the compiler-produced implementation. While prior approaches to HLS for ASICs and FPGAs have sometimes resulted in poor qual-

```
#define ROTL(x, b) (uint32_t) ((x << b) | (x >> (32 - b)))
void siphash(uint32_t v0, uint32_t v1, uint32_t v2, uint32_t v3) {
    1) v0 += v1;                    8) v0 += v3;
    2 ) v1 = ROTL(v1, 5);           9) v3 = ROTL(v3, 7);
    3) v1 ^= v0;                    10) v3 ^= v0;
    4) v0 = ROTL(v0, 16);           11) v2 += v1;
    5) v2 += v3;                    12) v1 = ROTL(v1, 13);
    6) v3 = ROTL(v3, 8);            13) v1 ^= v2;
    7) v3 ^= v2;                    14) v2 = ROTL(v2, 16);
}
```

Figure 4.2: Motivating Example ME-1: SipHash was manually split into four stages and rewritten by P4 programmers [155].

ity of the generated implementation, we believe that the narrower domain of packet-processing pipelines is particularly well-suited for applying HLS gainfully for 2 reasons. First, HLS techniques are designed to systematically explore tradeoffs between functionality (e.g., which ALU can implement an operation?), capacity (e.g., how many ALUs, gateways, etc.?), and scheduling of resources (which stage should run an operation?)—a core challenge in compiling to packet-processing pipelines. Second, HLS techniques can be effective in pipelining transactional code with updates to state, while providing transactional semantics to the programmer: the illusion that each packet modifies headers and state in isolation from other packets.

CaT overview. To produce high quality implementations, CaT combines ideas from several prior P4 compiler projects (Table 4.1) into an end-to-end system for the first time. CaT divides up the process of compiling a P4-16 program into three phases, as shown in Figure 4.1; the detailed relationship of these phases to prior HLS and compiler research is shown in Table 4.2. First, *resource transformations* rewrite packet-processing programs in P4 from one form (that makes use of a scarce resource) to another form (that makes use of a more abundant resource). Second, *re-*

source synthesis employs a novel algorithm based on program synthesis to synthesize low-level resource graphs with hardware ALUs, from a high-level transactional specification of a match-action table's action functionality. Third, *resource allocation* employs ILP or SMT solvers to allocate computations and data structures to memory blocks and action units, while respecting program dependencies and per-stage resource constraints. Throughout the 3 phases, CaT makes pervasive use of solver engines to simplify the development of and improve the quality of the compiler.

How CaT factorizes the compilation problem. The problem of optimal code generation in compilers is known to be NP-complete [7] in general. Within the context of P4, Vass et al. [149] show that the problem of compiling P4 programs to pipelines is NP-hard. These results suggest that a compiler may have to decompose the problem in some way to tradeoff optimality for reasonable performance. In CaT's approach, Phases 2 and 3 can be viewed as an *action-block based decomposition* of the P4 compilation problem. In Phase 2, we perform *local* resource synthesis for each individual action block (after transformations in Phase 1). Then, in Phase 3, we use these local synthesis results to perform a *global* resource allocation for *all* action blocks. This keeps the synthesis runtime manageable in practice while still attempting a good quality allocation of computation to resource units. Furthermore, our Phase 2 supports rich computations in action blocks that could require multiple stages as well as transactional (@atomic) semantics. In the rest of the paper, we refer to action block computations as *transactions*. The next three sections detail the three phases of our compiler.

4.4 Phase 1: Resource Transformation

In the first phase of our compiler, we perform source-to-source rewrites in P4, with the goal of transforming a program that makes use of scarce resources, to one that makes use of more abundant resources. Rewrite rules provide a flexible and general mechanism for this purpose,

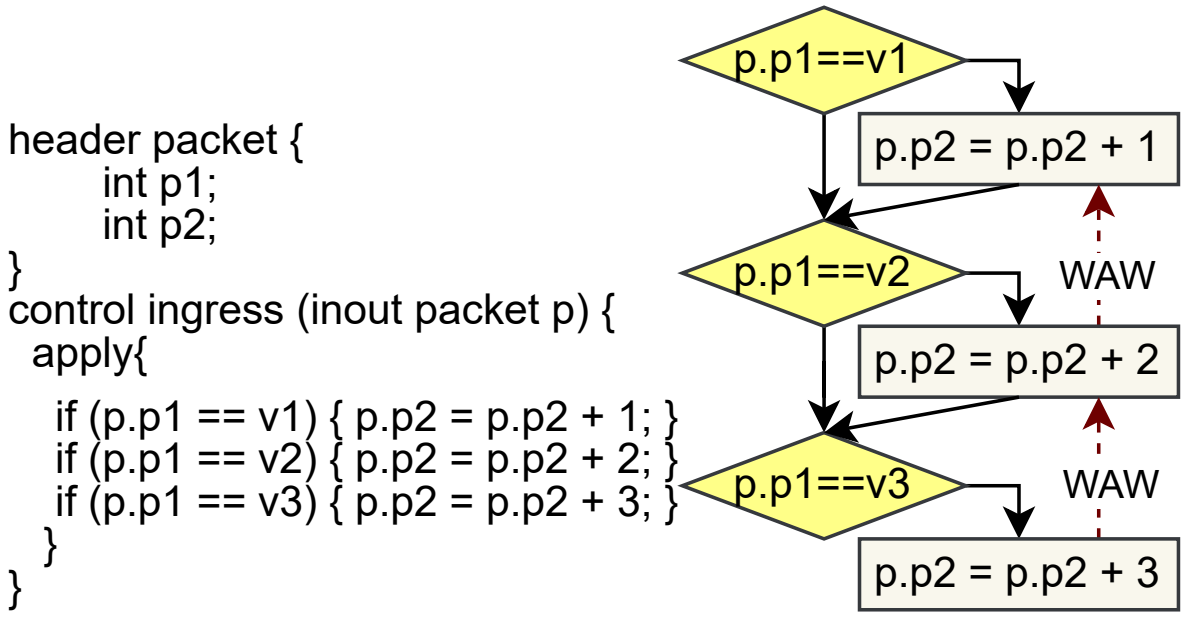

Figure 4.3: Motivating Example ME-2: The control flow graph of a P4 control block (left); snippet taken from a different portion of SipHash [129]. Dependencies shown as dotted red edges. $v1 \neq v2 \neq v3$ are constants.

and can be easily extended by adding more rules for new backend targets and resources. CaT includes rewrite rules for if-else statements in the control block of a P4 program. The standard p4c compiler transforms each action in an if-else branch into one default table, i.e., a table without a match key and with only one action. Our rewrites effectively merge together multiple (possibly nested) if-else statements into one bigger table with keys, thereby using fewer gateway resources (which are used to implement if-else branches). Such rewrites are in turn driven by a novel *guarded dependency analysis* that identifies parallelism opportunities by eliminating false dependencies — this often leads to reduced usage of pipeline stages in a program.

4.4.1 Guarded Dependencies

The sequence of program statements inside the `apply {...}` block of a P4 control block can be treated as a branching program (without loops) with (possibly nested) if-statements, reads and writes to PHV fields, and `apply` statements, which apply match-action tables. This program in-

duces Read-after-Write (RAW), Write-after-Read (WAR), and Writer-after-Write (WAW) dependencies between pairs of program statements, which must be respected during synthesis and resource allocation. Conventionally, these dependencies are defined between pairs of program statements without accounting for *path conditions* [79], i.e., conditions under which a control path in a program is executed. Specifically, a dependency due to variable v between two statements s_1 and s_2 is denoted as $(v@s_1 \rightarrow v@s_2, t)$, where $t \in \{RAW, WAR, WAW\}$.

Consider the motivating example ME-2 shown in Figure 4.3, inspired by a different portion of the SipHash program [129]. The WAW dependencies (shown on the right) cause the Tofino compiler to produce an implementation with 3 pipeline stages. However, *these WAW dependencies are not real*, since the if-conditions guarding these assignments are disjoint. Indeed, a developer of this program recognized the disjoint conditions and *manually changed* the program to use a single block of `if...else if...else if...` statements, thereby reducing the pipeline usage of the compiled program to 1 stage. *We aim to automate such rewrites.* In particular, p4c and the Tofino compiler miss these rewrites in ME-2, likely due to a conservative dependency analysis.

To solve this issue, we propose *guarded dependencies*, which take into account path conditions along control paths. Given a control-flow graph (CFG) C for a P4 control block, a *guarded dependency* between nodes $(n_1, n_2) \in C$ is defined as a tuple $(v@s_1 \rightarrow v@s_2, t, \phi)$, where v is the variable of concern at statement s_1 (in node n_1) and statement s_2 (in node n_2), $t \in \{RAW, WAR, WAW\}$, and ϕ (called a guard) is a formula that describes all the path conditions under which node n_2 may be visited after node n_1 is visited. A procedure based on symbolic execution [79] or model checking [25] that computes path conditions can be used to determine precise guarded dependencies for the program. In particular, we can use an SMT solver to identify *false dependencies*, i.e., dependencies where ϕ is unsatisfiable. Since running model checking or symbolic execution on the input program can be expensive, we next describe a faster lightweight analysis for analyzing guarded dependencies in CaT.

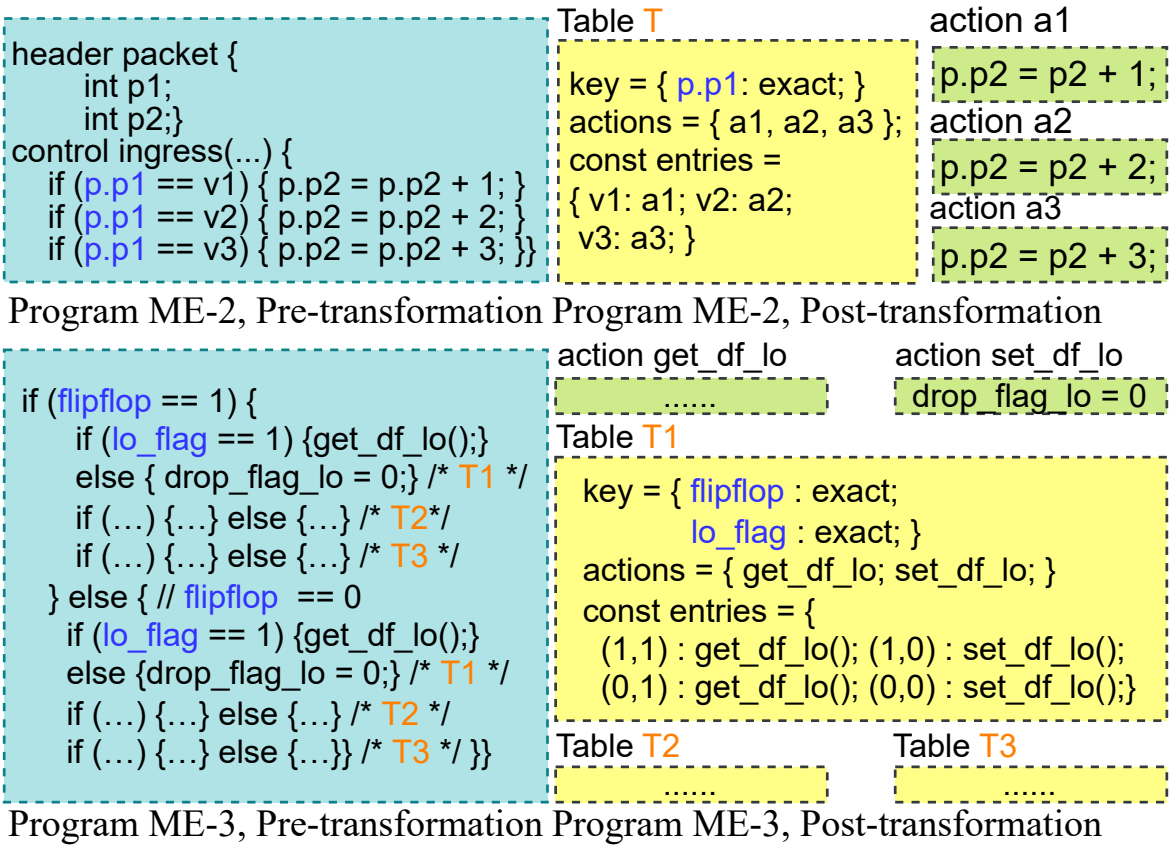

Figure 4.4: Illustration of Phase 1 Rewrites in CaT, on motivating examples ME-2 (from Figure 4.3) and ME-3 (from a UPF Rate_enforcer [117] example provided by P4 programmers).

4.4.2 Lightweight Guarded Dependency Analysis for CaT Rewrites

We now describe a lightweight analysis that helps CaT determine *guarded dependencies* in a P4 program. First, we check that none of the assignment statements update any variables used in conditions of if-else statements. Such updates lead to WAR dependencies and complicates the analysis; we currently choose to not perform any rewrites in such cases. When there are no such updates, the path condition for each CFG node is a simple logical AND of branch conditions, which we compute by a depth-first traversal of the CFG. During the depth-first traversal, branch conditions are pushed/popped on a stack at branch/merge points, respectively. For a pair of

nodes $(n_1, n_2) \in C$ with a guarded dependency, the guard ϕ is a logical AND of the computed path conditions for n_1 and n_2. If ϕ is unsatisfiable, then this is a false dependency and removed; otherwise it is conservatively retained as a dependency. For our ME-2 example in Figure 4.3, this analysis finds that the shown WAW dependencies are false, and removes them.

4.4.3 Rewrites to Match-Action Tables

We now focus on (possibly nested) if-else statements where the branch conditions are tests on packet fields that can be implemented as keys in a match-action table. Based on the guarded dependency analysis, if there is no dependency between the branches, then we can rewrite them into a match-action table. The key of the generated table is comprised from packet fields used in the if-else conditions, and the actions are the computations within each branch. For example, Figure 4.4 illustrates our rewrites on two P4 programs – ME-2, and another motivating example ME-3 taken from a UPF Rate_enforcer example [89]. After rewriting, both ME-2 and ME-3 use only match-action tables and thus no gateway resources. ME-2 uses only 1 stage post-rewriting vs. 3 stages before rewriting (due to false WAW dependencies). ME-3 also uses only 1 stage post-rewriting vs. 2 stages before rewriting (due to needing too many gateway resources to fit into 1 stage). These motivating examples drawn from real-world P4 programs show the effectiveness of our approach, where *manual steps taken by a programmer to reduce resource usage are successfully automated by CaT.*

4.5 Phase 2: Resource Synthesis

For the second phase, we propose a novel procedure to perform resource synthesis on each P4 action block. Like Chipmunk [50], we too use the SKETCH program synthesis tool [132] to generate a *semantically equivalent* pipelined hardware implementation using ALUs. However, there are several important differences from Chipmunk, which we summarize at the end of this

Input:
1. Computation graph $G = (V, E)$, with $V = \mathsf{Stateful} \cup \mathsf{Stateless}$, where Stateful is the set of stateful nodes and Stateless is the set of stateless nodes;
2. Primary outputs POs: Outgoing edges of G
3. Stateful ALU grammar A_1, stateless ALU grammar A_2;
4. Number of pipeline stages available in hardware, `numPipelineStages`.

Output: Synthesized code for each primary output (PO) and each stateful update.

```
1  // Step 1: Normalize the computation graph to ensure every stateful
     node in G has out-degree 1.
2  Normalize(G);
3  // Step 2: Perform predecessor packing and folding optimizations.
4  graphModified ← TRUE;
5  // Iterate until fixpoint
6  while graphModified do
7      // Folding: tryFold returns TRUE iff G changed
8      for (u, v) ∈ E do
9          if v ∈ Stateful ∧ u ∈ Stateless then
10             graphModified ← tryFold (G, u, v, A1);
11         end
12     end
13     // Predecessor packing: tryPack returns TRUE iff G changed
14     for (u, v) ∈ E do
15         if u ∈ Stateful ∨ v ∈ Stateful then
16             graphModified ← tryPack (G, u, v, A1);
17         end
18     end
19 end
20 // Step 3: Synthesis of stateful updates
21 for v ∈ Stateful do
22     s ← querySketchStateful (v, A1);
23     if s ≡ FAILURE then
24         abort(
25         "Error synthesizing stateful node " + v);
26     end
27 end
28 // Step 4: Min-depth solutions for stateless code
29 Os ← POs ∪ {inputs(v)|v ∈ Stateful};
30 Order elements of Os according to topological order in G;
31 for o ∈ Os do
32     // Compute the Backwards Cone of Influence (BCI) of o
33     spec ← computeBCI(o); // spec of o
34     i ← 1; // initial depth of solution tree
35     // Loop over i to find a minimum depth solution tree
36     while i ≤ numPipelineStages do
37         s ← querySketchStateless(spec, i, A2);
38         if s ≡ SUCCESS then
39             break;
40         else
41             i ← i + 1; // increment depth
42         end
43     end
44 end
```

Auxilliary procedures:

procedure tryFold(G, u, v, A_1): Query SKETCH to determine if edge (u, v) can be folded into stateful node v using stateful grammar A_1. If query succeeds, edge (u, v) is removed from G.

procedure tryPack(G, u, v, A_1): Query SKETCH to determine if nodes u, v can be packed into a single new stateful node using stateful grammar A_1.

Figure 4.5: CaT Synthesis Procedure. Calls that use a SKETCH query are highlighted in blue box.

section, after describing our procedure.

4.5.1 Preprocessing of a P4 Action

We preprocess each action block of the P4 program to prepare for synthesis. We first use some standard preprocessing steps, similar to Domino [130], including (a) branch removal (by replacing assignments under branches with conditional assignments), (b) creating two temporary packet fields for each stateful variable – *pre-state field* (denoting its value before update) and *post-state field* (denoting its value after update), and (c) conversion to static single-assignment (SSA) form [38].

In addition, and differently from Domino, we perform several static analyses during prepro-

cessing: constant folding, expression simplification, and dead code elimination. These analyses are useful in simplifying the action block of a P4 program, thereby reducing the difficulty of the subsequent SKETCH queries. While preprocessing can create temporary packet fields, we neither add nor delete stateful variables during preprocessing simplifications.

4.5.2 Computation Graph for a P4 Action

After preprocessing, we construct a dependency graph (similar to Domino), with nodes for each program statement and an edge for each RAW dependency.[2] Edges in both directions are also added to/from the pre/post-state fields of each stateful variable. The strongly connected components (SCCs) of this graph correspond to stateful updates, which are condensed to form a *computation graph* G. Thus, G is a directed acyclic graph (DAG) with nodes for program computations (some with stateful updates) and edges for RAW dependencies. Nodes in G are partitioned into two sets: *stateful nodes* are formed from SCCs on the dependency graph, containing a set of program statements that describe an atomic stateful update; *stateless nodes* are the other nodes in the dependency graph. Each edge (u, v) is mapped to a packet field variable that appears in the LHS of the assignment at u and in the RHS of the assignment at v. We call source edges of G *primary inputs (PIs)*; each is associated with an input packet field variable. We call outgoing edges of G *primary outputs (POs)*, each is associated with a final value written to a packet field variable.

4.5.3 Synthesis Procedure for a P4 Action

Synthesis for a P4 action is now performed on the computation graph G. Instead of creating a large synthesis query for the entire G, we decompose the problem into multiple smaller synthesis queries. Specifically, we generate individual synthesis queries for the following variables in G: (1) the output stateful variable of each stateful node (i.e., the LHS of the stateful update

[2] Conversion to SSA form removes WAW and WAR dependencies.

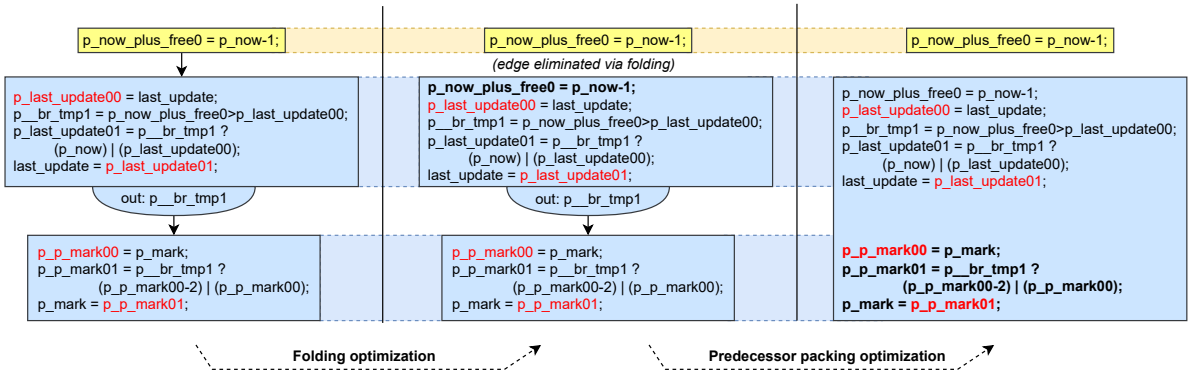

Figure 4.6: Computation graph for the BLUE(decrease) [45] (leftmost) and optimizations performed by CaT when targeting the Tofino ALU. Stateful nodes are in blue, stateless nodes are in yellow, pre/post-state fields are in red, modified parts are in bold.

assignment), (2) each input variable to a stateful node (i.e., any variable in the RHS of a stateful update assignment), and (3) each primary output (PO) variable, which corresponds to a packet field. Each synthesis query finds an ALU-based implementation and is *parameterized* by an ALU grammar that specifies the functionality of the ALUs (stateful or stateless) available in a given hardware target. These implementations are then connected together according to G, to result in a *resource graph* R, where a node v represents an ALU, and an edge (u, v) indicates that the output of ALU u is connected to an input of ALU v. We prove that our synthesis procedure is correct: the resource graph R is functionally equivalent to G.

Our synthesis procedure is shown in **Algorithm 1**, which consists of four main steps: 1) normalization; 2) folding and predecessor packing optimizations; 3) synthesis of stateful updates; 4) synthesis of minimum-depth solutions for stateless code. The critical step is Step 3, which queries SKETCH to see if each stateful update assignment can be synthesized into configurations for a single stateful ALU. If any such query fails, then we terminate the procedure and provide feedback to the programmer. We create separate synthesis queries to perform optimizations in Step 2, to help Step 3 succeed. Finally, Step 4 creates synthesis queries to implement the POs and inputs to the stateful nodes.

Step 1: Normalization of computation graph G In the typical hardware backends that we target

(e.g., Menshen, Tofino), a stateful ALU can output a single value that is either the pre-state or the post-state value of one of its stateful registers. In this step, we normalize G to a graph such that each stateful node has only one output, and each packet field labelled as an out-edge from a stateful node is either the *pre-state field* or the *post-state field*. Normalization is performed by replicating stateful nodes that have multiple outputs.

Step 2: Folding and predecessor packing optimizations We iterate the following two optimizations until convergence.

Folding to reduce input edges. A stateful node with too many in-edges could cause Step 3 to fail, due to a limited number of inputs available in ALUs. The **folding optimization** finds opportunities to reduce the number of in-edges to a stateful node. We consider *dependent inputs*, i.e., inputs that are themselves functions of other inputs to the same stateful node. For each such candidate i, we query SKETCH to check if the function that computes i can be *folded into* the stateful node itself, such that the enlarged node fits into a single stateful ALU. If the synthesis query is successful, i is removed. Figure 4.6 shows an example benchmark—BLUE (decrease) [45]—where this works successfully. Here, *folding* reduces an edge between the top two nodes in the computation graph G (extreme left of Figure 4.6), thereby reducing the pipeline usage by 1.

Predecessor packing to merge nodes. Even after folding, the stateful update in a single node in G might not *fully utilize* an available stateful ALU in hardware. Consider again the BLUE (decrease) example in Figure 4.6, where the middle box shows G after folding. Here, a single Tofino stateful ALU can actually implement *both* stateful updates (in blue boxes) in a single stage, as shown by a merged node on the right. To achieve this compaction, we use a simple heuristic called **predecessor packing**, inspired by technology mapping for hardware designs [36]. The key idea is to pack more into a stateful ALU by attempting a merge of nodes u and v, where at least one node is stateful and where predecessor u has only one out-edge (to v). Like folding, we implement the packing attempt via a SKETCH query, and merge the nodes if the query is successful. In our evaluations (§4.7.3), we show that these optimizations are effective in compiling to fewer pipeline

stages.

Step 3: Synthesizing stateful updates We are now ready to synthesize the outputs of the stateful nodes in G. To preserve the transactional semantics of the program, each stateful update must be completed within a single pipeline stage, i.e., *the update operation must fit in a single stateful ALU.* Accordingly, for each stateful node in G, we generate a SKETCH query to check if the stateful update operation can be implemented by a single stateful ALU. The functionality of the stateful ALU available in hardware is specified using an ALU grammar A_1, which is expressed as a large block of multiple if-else statements with one case for each opcode. We assert that each such query succeeds; if any query fails, our procedure exits with an error, giving feedback to the programmer.

Step 4: Minimum-depth solutions for stateless code In the last step, we synthesize code for the POs and inputs to the stateful nodes in G (line 29). For each such variable o to be synthesized, we first compute its backwards cone of influence (BCI), which is often used in verification/synthesis tasks to determine the dependency region up to some (boundary of) inputs [57]. In graph-theoretic terms, $BCI_G(o)$ is a subgraph in G derived by going recursively backward from o, stopping at a PI or an output of a stateful node. Essentially, the BCI provides the *functional specification* for o in terms of a set of inputs, where each input is a PI or the output of a stateful node in G. Note that these specifications are stateless, i.e., they do not include any stateful nodes.

We model a switch's stateless ALU functionality using an ALU grammar A_2 (expressed as a large block of if-else statements). We use SKETCH to find a *minimum-depth tree* solution for o, where each tree node represents a stateless ALU, and the leaf nodes represent the inputs in $BCI_G(o)$. A minimum-depth solution helps reduce the number of pipeline stages – this is explained in more detail in the next section (§4.5.4). Since SKETCH does not support optimal synthesis, we invoke it in a loop to minimize depth, where each iteration tries to find a solution tree of a given depth i (line 35), starting from 1 and continuing until i exceeds the maximum number of pipeline stages. An example computation graph with a single stateful update (blue box) and the associated synthesis query results are shown in Figure 4.7.

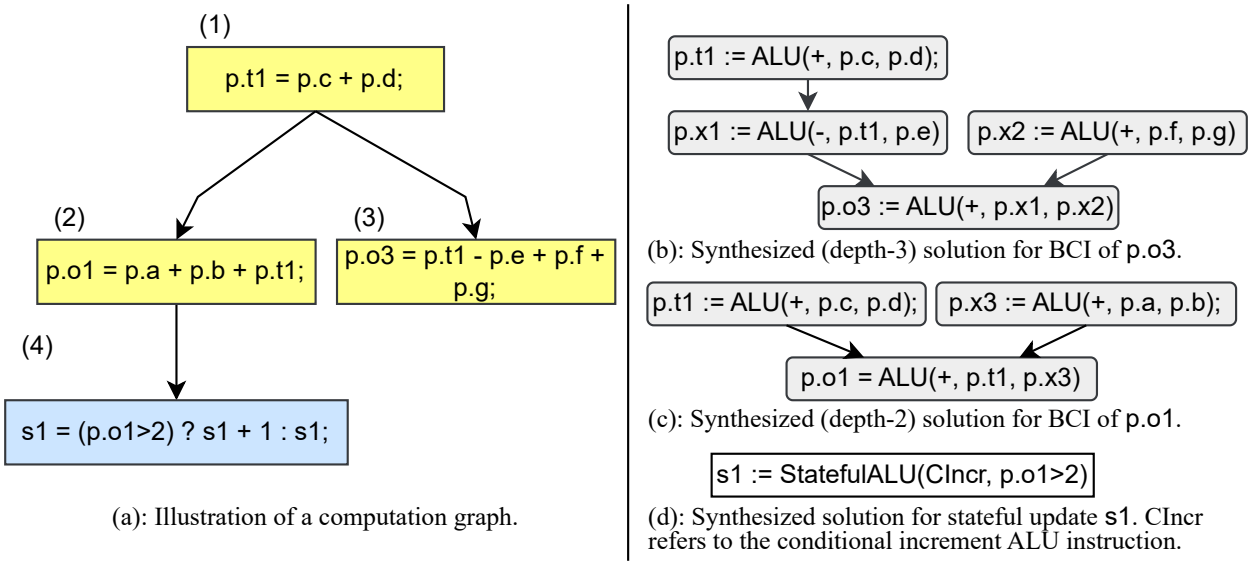

Figure 4.7: Example of a computation graph (left) and the synthesis query results (right) targeting Banzai ALUs [130]. Stateful nodes in blue and stateless nodes in yellow. The POs are: {p.o1,p.o3}. p.o1's BCI contains nodes 1 and 2; p.o3's BCI contains nodes 1 and 3.

4.5.4 Staged-Input Tree Grammar for Synthesis

We now describe details of the grammar used for the synthesis queries in Step 4, where each query (in line 37) tries to find a solution tree of a given depth i for implementing a given variable o. Initially, we used a simple recursive tree grammar for the SKETCH query, where each tree node is an ALU (specified by a stateless ALU grammar A_2) and its children are the ALU operands; and a leaf node is an input in $BCI_G(o)$, i.e., either a primary input (PI) or an output of a stateful node in G. By iteratively incrementing i, we were able to find a minimum-depth tree solution for o.

However, even with a minimum-depth tree solution for each variable o, when we compose together these solution trees according to G, the number of pipeline stages for the entire action may not be the minimum possible. This is because with this simple grammar, the depth is optimized to be minimum *within an individual synthesis query for* o, without considering the larger scope of the entire action. As a concrete example, consider the computation graph G for the

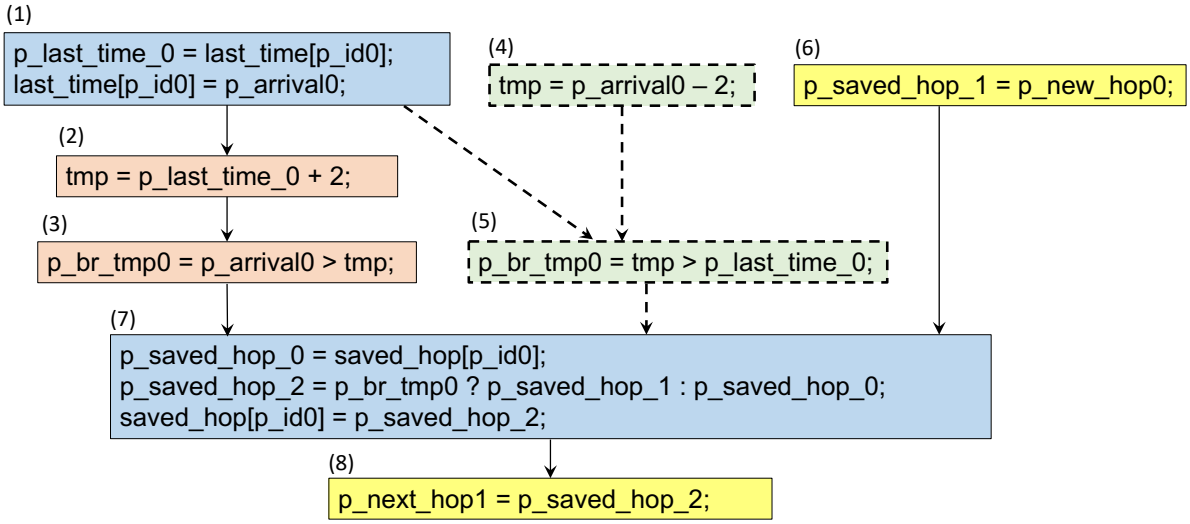

Figure 4.8: Computation graph for the Flowlet switching [128], showing two minimum-depth solutions for variable `p_br_tmp0` using Banzai ALUs: nodes {2, 3} and nodes {4, 5}.

Flowlet switching benchmark [128] shown in Figure 4.8. As before, blue nodes are stateful nodes and yellow nodes are stateless. In addition, we show two synthesized solutions for the variable `p_br_tmp0`, with the specification `p_br_tmp0 = (p_arrival0 - p_last_time_0 > 2)`. Its BCI has two inputs: `p_arrival0` is a PI, and `p_last_time_0` is the output of the stateful node 1.

Note first that the BCI input `p_last_time_0` can only be available after stage 1 within the overall action (stages are numbered starting from 1), since the implementation of node 1 occupies one stage. Now, consider a minimum-depth solution with nodes 2 and 3 (both shown in orange), where node 1 provides an input to the ALU operation in node 2, which in turn provides an input to the ALU operation in node 3, which computes `p_br_tmp0`. Hence, `p_br_tmp0` is computed in stage 3 and is available at the end of stage 3.

Consider a second minimum-depth solution shown by dashed nodes and edges, with nodes 4 and 5 (both shown in green). Like the first solution, it also has two ALU operations and the same minimum depth 2. However, the ALU operation in node 4 is *independent* of `p_last_time_0`, and can be computed in parallel with node 1. This allows `p_br_tmp0` to be computed in stage 2, making it available at the end of stage 2. This example shows that although both solutions have

the minimum depth 2, the second is better because `p_br_tmp0` can be computed in an earlier stage for the overall action.

Since the number of pipeline stages is often a critical resource in compiling P4 programs, we consider the larger scope of the action in each individual synthesis query. We achieve this by augmenting our tree grammar for a query, where an input in the BCI is now associated with a *stage*, which denotes the stage *within the action* at which the input is available to be used. We call this grammar a *staged-input tree grammar*. We regard a primary input (PI) in G as being available for use at stage 1, and the output of a stateful node being available for use at some stage $s > 1$, where s depends on its own implementation. In each individual synthesis query, we now look for a minimum-depth tree solution that produces the output at the *earliest possible stage*, based on stage information of the inputs in its BCI. To compute the latter, in Step 4, we use a topological ordering over the set of outputs o in G (line 30), such that any input in $BCI_G(o)$ is already implemented before the synthesis query for o. For the example in Figure 4.8, our synthesis query with a staged-input tree grammar returns the solution with nodes 4 and 5 (in green) for the output `p_br_tmp0`. The complete SKETCH input for this query is shown in Figure 4.9, which includes the grammars for a staged-input tree and for a stateless ALU.

4.5.5 Final Result of the Synthesis Procedure

The final result of the synthesis procedure is represented in the form of a resource graph R for a given P4 action block, where each node v in R represents a stateful or a stateless ALU, and an edge (u, v) in R indicates that the output of ALU u is connected to an input of ALU v. These resource graphs play an important role in resource allocation, the next phase of our compiler. We now state and prove correctness of our synthesis procedure.

Theorem 1 (Correctness). *The result of the CaT synthesis procedure (Algorithm 1) on a computation graph G is correct.*

```
// SKETCH input file for one synthesis query
// ALU grammar specification
int alu(int opcode, int pkt_0, int pkt_1, int pkt_2, int immediate_operand) {
    if (opcode == 0) {
        return immediate_operand;
    } else if (opcode == 1) {
        return pkt_0 + pkt_1;
    } else if (opcode == 2) {
        return pkt_0 + immediate_operand;
    } else if (opcode == 3) {
        return pkt_0 - pkt_1;
    } else if (opcode == 4) {
        return pkt_0 - immediate_operand;
    } else if (opcode == 5) {
        return immediate_operand - pkt_0;
    } else if (opcode == 6) {
        return pkt_0!=pkt_1;
    } else if (opcode == 7) {
        return (pkt_0 != immediate_operand);
    } else if (opcode == 8) {
        return (pkt_0 == pkt_1);
    } else if (opcode == 9) {
        return (pkt_0 == immediate_operand);
    } else if (opcode == 10) {
        return (pkt_0 >= pkt_1);
    } else if (opcode == 11) {
        return (pkt_0 >= immediate_operand);
    } else if (opcode == 12) {
        return (pkt_0 < pkt_1);
    } else if (opcode == 13) {
        return (pkt_0 < immediate_operand);
    } else if (opcode == 14) {
        return pkt_0 != 0 ? pkt_1 : pkt_2;
    } else if (opcode == 15) {
        return pkt_0 != 0 ? pkt_1 : immediate_operand;
    } else if (opcode == 16) {
        return ((pkt_0 != 0) || (pkt_1 != 0));
    } else if (opcode == 17) {
        return ((pkt_0 != 0) || (immediate_operand != 0));
    } else if (opcode == 18) {
        return ((pkt_0 != 0) && (pkt_1 != 0));
    } else if (opcode == 19) {
        return ((pkt_0 != 0) && (immediate_operand != 0));
    } else {
        return (pkt_0 == 0);
    }
}
// staged-input tree grammar for implementation (vars0, vars1, and vars are specific to each query and are defined in the harness
     below)
generator int expr(fun vars0, fun vars1, fun vars, int bnd){
    if (bnd == 0){
        return vars0();
    }
    int t = ??(1);
    if (t == 0) {
        return vars();
    }
    else {
        return alu(??, expr(vars0, vars1, vars, bnd-1), expr(vars0, vars1, vars, bnd-1), expr(vars0, vars1, vars, bnd-1), ??);
    }
}
// specification function, with BCI inputs as arguments
int comp_5(int pkt_arrival, int pkt_last_time00) {
    bit pkt__br_tmp1;
    pkt__br_tmp1 = pkt_arrival-pkt_last_time00>5;
    return pkt__br_tmp1;
}
// harness for synthesis
harness void sketch(int pkt_arrival, int pkt_last_time00) {
    generator int vars0(){
        return {| pkt_arrival |};
    }
    generator int vars1(){
        return {|pkt_last_time00|};
    }
    generator int vars(){
        return {| pkt_arrival | pkt_last_time00|};
    }
    // synthesized expression must be equivalent to specification
    assert expr(vars0, vars1, vars, 2) == comp_5(pkt_arrival, pkt_last_time00);
}
```

Figure 4.9: One example of the generated synthesis query for SKETCH.

Proof sketch. The synthesis procedure works by decomposing G (after correctness-preserving normalization and optimizations in Steps 1 and 2, respectively) into subgraph components com-

prising of: (1) outputs and inputs of stateful nodes, (2) inputs of stateful nodes and their stateless BCIs, and (3) POs and their stateless BCIs. Each such subgraph of G represents a *specification* for a synthesis query (in Steps 3 or 4), which generates a corresponding *implementation* using ALUs, i.e., a subgraph in the resource graph R. Based on correctness of program synthesis in SKETCH [132], each stateful node output, stateful node input, and PO in R is functionally equivalent to that in G. Hence the synthesis procedure is correct. □

4.5.6 Comparison with Synthesis in Chipmunk

Our motivation for a new synthesis procedure was improving the performance of synthesis in Chipmunk [50], which also uses SKETCH. CaT and Chipmunk have several differences.

First, CaT creates multiple *smaller* synthesis queries for SKETCH. Although Chipmunk uses a slicing technique to create per-output queries, the scope for each such query is the entire transaction. Our procedure separates queries for stateful update operations from those on stateless operations in Steps 3 and 4, respectively. The scope for a stateful query is a single stateful ALU: these queries are small and also critical; if any fails, synthesis cannot succeed. The scope for a stateless query is typically smaller than an entire transaction, since its BCI stops at outputs of other stateful nodes. Overall, smaller synthesis queries lead to significant performance improvement over Chipmunk, as demonstrated in evaluations (§4.7). We note that because SKETCH queries are independent of each other in both CaT and Chipmunk, both lose opportunities to share common computations across multiple queries.

Second, for stateless operations, we use multiple SKETCH queries to synthesize solutions of *minimum-depth*, i.e., the minimum number of pipeline stages, while searching the space over all possible equivalent programs. Although Chipmunk also considers the space of all possible programs, its queries do not guarantee minimum-depth solutions within a given bound.

Third, Chipmunk creates synthesis queries in the form of low-level holes in an ALU grid architecture that are filled by SKETCH. In contrast, our synthesis queries ask for ALU-based

implementations that we represent as resource graphs. These resource graphs are used during resource allocation (in Phase 3) for handling *multiple* transactions, which are not supported by Chipmunk.

Finally, similar to Chipmunk's ALU DSL, our synthesis queries are *parameterized* by an ALU grammar that specifies the functionality of ALUs available in a given hardware target. This enables the same synthesis procedure to be used for different hardware backends, providing compiler retargetability. CaT currently supports three different ALU grammars: Tofino ALUs [68], Banzai ALUs [130], and Menshen ALUs [150]; more can be supported as needed. As long as compiler developers have access to the documentation of a hardware ALU in the target backend, it is straight-forward to write a complete and correct ALU grammar describing its capabilities.

4.6 Phase 3: Resource Allocation

After performing synthesis for each P4 action block, the third phase of our compiler performs *global* resource allocation for the full P4 program by using a constraint-based formulation, shown in Table 4.3. The top part lists the definitions of constants, indices, variables, and sets that are used to automatically generate the constraints. The bottom part shows the full set of constraints, divided into a first set that is similar to prior work [75, 86], and a second set that is new. Our new constraints address: (1) ALU resources in action computations, (2) multi-stage actions, (3) fitting multiple action blocks in the same pipeline stage, and (4) propagation of ALU outputs. Prior efforts either do not consider allocation of ALU resources and multi-stage actions [75, 86], or do not address multiple action blocks [50, 130]. Another novel feature of our approach is that we use the resource graph R synthesized for each action block (in Phase 2), to perform global optimization in this phase.

	Name	Definition
Constants	N_S	maximum number of pipeline stages
	N_{alu}	maximum number of ALUs in each stage
	N_P	number of packet fields that must remain available through the pipeline
	N_{table}	maximum number of logical table IDs per stage
	$N_{entries}$	maximum number of match entries per table per stage
	e_t	maximum number of entries in table t in program
Indices	t	index for Table
	i	index for partition of a Table, partition denoted $t[i]$
	a	index for Action
	u	index for ALU
	s	index for pipeline Stage
Variables	M_{tis}	binary, set to 1 iff match of $t[i]$ is assigned to stage s, 0 otherwise
	$stage_u$	integer, stage assigned to ALU u
	$stage_{us}$	binary, set to 1 iff ALU u is assigned to stage s
	beg_u	integer, stage where ALU u output is computed
	end_u	integer, $\geq$ last stage where ALU u is used as an input
	$prop_{us}$	binary, set to 1 iff output of ALU u is propagated in stage s
Sets	R_{tia}	Resource graph for action a of table partition $t[i]$
	V_{tia}	Vertices in R_{tia}, each represents an ALU
	E_{tia}	Edges in R_{tia}, each represents a connection between ALUs
	AP_{tia}	Set of ALUs in R_{tia}, whose outputs may need to be propagated across stages
	UV_{tiau}	Set of ALUs v in R_{tia}, s.t. $(u, v) \in E_{tia}$, i.e., ALU u is an input to ALU v
Constraints similar to prior work [75, 86]		
Match table capacity	$\forall s: \ \sum_{t,i} \ M_{tis} \leq N_{table}$	
Match action pairing	$\forall s \ \forall t, i, a \ \forall u \in V_{tia}: \ \ stage_{us} \rightarrow M_{tis}$	
Table dependency	$\forall i_1, i_2, a_1, a_2, \forall u_1 \in V_{t_1 \ i_1 \ a_1}, \ \forall u_2 \in V_{t_2 \ i_2 \ a_2} : stage_{u_1} < stage_{u_2}$	
New constraints in our work		
ALU allocation 1	$\forall t, i, a \ \forall u \in V_{tia}: \ \ 1 \leq stage_u \leq N_S$	
ALU allocation 2	$\forall s \ \forall t, i, a, \ \forall u \in V_{tia}: \ \ stage_u = s \leftrightarrow stage_{us}$	
Action dependency	$\forall t, i, a \ \forall (u, v) \in E_{tia}: \ \ stage_u < stage_v$	
ALU propagation 1	$\forall t, i, a \ \forall u \in AP_{tia}: \ \ beg_u = stage_u \wedge beg_u < end_u \wedge end_u \leq N_S$	
ALU propagation 2	$\forall t, i, a \ \forall u \in AP_{tia}, \forall v \in UV_{tiau}: \ \ end_u \geq stage_v$	
ALU propagation 3	$\forall t, i, a \ \forall u \in AP_{tia}, \forall s \in \{1, \ldots, N_S\}: \ \ (beg_u < s \wedge s < end_u) \leftrightarrow prop_{us}$	
ALU propagation 4	$\forall s \ \sum_{\forall t,i,a \ \forall u \in AP_{tia}} (stage_{us} + prop_{us}) \leq N_{alu} - N_P$	

Table 4.3: Constraint formulation for resource allocation.

4.6.1 Constraints Similar to Prior Work

If a match table in the program has too many entries to fit within a single stage, it is partitioned into b_t separate tables, where $b_t = \lceil (e_t / N_{entries}) \rceil$. Currently, we only support exact matches; hence, a packet will match at most one of the partitions $t[i]$ that have the same actions as table t. The first constraint ensures that the number of match tables allocated in a stage is less than or equal to the number of table IDs available. The second ensures that ALUs in action blocks are accompanied by the associated match table. The third enforces four types of table dependencies: match, action, successor, and reverse-match [75]. If table t_2 depends on table t_1, all ALUs of t_2 are allocated after ALUs of $t1$. For successor and reverse-match, $<$ is replaced by $\leq$.

4.6.2 New Constraints in Our Work

The constraints for ALU allocation (1,2) ensure that each ALU in each action is assigned to one and only one pipeline stage. The Action dependency constraint uses the edges in R_{tia} (synthesized in Phase 2) to enforce dependencies between ALUs. Together with the Table dependency constraint, this allows ALUs from multiple action blocks to be assigned in the same pipeline stage, *while respecting both inter-table and intra-action dependencies.*

We support a multi-stage action under the condition that it does not modify the table's match key, by duplicating the match entries at each stage to ensure that the entire action is executed. As an example, suppose a match entry m in table t is associated with action A that takes 2 stages. We can allocate table t in two consecutive stages, such that if a packet matches entry m in table t in stage s, it will match entry m in table t in stage $s+1$ as well, resulting in action A being executed completely over the two stages.

We allow allocation of multiple actions in the same stage and also allow assigning an Action A to *non-consecutive* stages. In the latter case, we need additional ALUs in the intermediate stages to *propagate the intermediate results.* The ALU propagation constraints (1-4) handle allocation

of these additional ALUs. Here, AP_{tia} is the set of ALUs in R_{tia} whose outputs may need to be propagated across stages, and UV_{utia} is a set of ALUs v in R_{tia} that use ALU u as an input. The ALU propagation constraints 1-3 ensure that an ALU $u \in AP_{tia}$ is propagated until the largest stage where it is used as an input. The ALU propagation constraint 4 enforces the ALU capacity constraint in each stage, where N_P ALUs are pre-occupied to carry packet fields that remain live through the whole pipeline (e.g., IP TTL) or are updated (by ALUs not in any AP_{tia}); the remaining $(N_{alu} - N_P)$ ALUs must be enough for the sum over all ALUs u in any AP_{tia} that are either assigned to or propagated in that stage.

4.6.3 ILP encoding for ALU propagation constraints

We use the big-M method to obtain an ILP formulation of the constraint

$$\forall u \in I, \forall s \ (beg_u < s \wedge s < end_u) \leftrightarrow prop_{us} = 1$$

For each $u \in I$ and $s \in \{1, \ldots N_S\}$, we use a binary variable lo_{us} as an indicator for $beg_u < s$ and a binary variable hi_{us} as an indicator for $s < end_u$. M is a large constant (e.g., $N_S + 5$).

The following constraints ensure that lo_{us} is 1 if $beg_u < s$ and 0 otherwise.

$$s - beg_u \leq M lo_{us} \tag{4.1}$$

$$s - beg_u > -M(1 - lo_{us}) \tag{4.2}$$

If $s - beg_u > 0$ then $lo_{us} = 1$ (4.1) and if $s - beg_u \leq 0$ then $lo_{us} = 0$ (4.2).

The following constraints ensure that hi_{us} is 1 if $s < end_u$ and 0 otherwise.

$$s - end_u < M(1 - hi_{us}) \tag{4.3}$$

$$s - end_u \geq -M hi_{us} \tag{4.4}$$

If $s - end_u < 0$ then $hi_{us} = 1$ (4.4) and if $s - end_u \geq 0$ then $hi_{us} = 0$ (4.3).

The following constraints use lo_{us} and hi_{us} to make $prop_{us}$ an indicator for $beg_u < s < end_u$.

$$lo_{us} + hi_{us} - 2 < Mprop_{us} \tag{4.5}$$

$$lo_{us} + hi_{us} - 2 \geq -M(1 - prop_{us}) \tag{4.6}$$

If $lo_{us} + hi_{us} - 2 \geq 0$ then $prop_{us} = 1$ (4.5) and if $lo_{us} + hi_{us} - 2 < 0$ then $prop_{us} = 0$ (4.6). This means that $prop_{us} = 1$ only if both $lo_{us} = 1$ and $hi_{us} = 1$. Hence, $prop_{us} = 1$ if $s > beg_u$ and $s < end_u$, otherwise $prop_{us} = 0$.

4.6.4 Solving the Constraint Problem

We can use either an ILP solver (Gurobi [56]) or an SMT solver (Z3 [146]) to find an optimal or a feasible solution. We specify an objective function to find an optimal solution, e.g., we add the constraint min $cost$ to minimize the number of stages, where $cost$ is $\geq$ the stage assigned to any ALU. i.e., $\forall t, i, a,\ \forall u \in V_{tia} :\ cost \geq stage_u$. To find a feasible solution, we use a trivial objective function ($min\ 1$) with Gurobi (none is needed with Z3).

4.7 Implementation and Evaluation

We implement the CaT compiler with the workflow shown in Figure 4.1. The resource transformation phase is implemented on top of p4c [100]. We also use p4c to identify the action blocks and table dependencies needed in CaT's resource synthesis and resource allocation phases. For the backend, ideally the CaT compiler should directly output machine code for the targets. However, due to the undocumented and proprietary machine code format of the Tofino chipset, we generate a low-level P4 program by using a best-effort encoding for the resource constraints, based on known information about the Tofino chipset. For the Menshen backend, we extend the

open-source RMT pipeline [94, 150] by writing additional Verilog to support richer ALUs, e.g., the IfElseRAW ALU [130]. The CaT compiler directly outputs machine code to configure various programmable knobs (e.g., opcodes) within Menshen's Verilog code.

Sanity checking of CaT prototype We check CaT's output for Menshen using its cycle-accurate simulator, which can be fed input packets to test the generated machine code. We create P4-16 benchmarks starting with a subset of the switch.p4 program [106], consisting of 2–6 tables randomly sampled from switch.p4. Then, we add new actions to the tables using @atomic blocks for transactional behavior. The logic within these atomic blocks consists of one of 8 Domino benchmark programs [130]; the IfElseRaw ALU [130] in our simulator is not expressive enough for the remaining 6. We also test the 8 benchmark programs in isolation, generating 24 benchmarks in total, many of which have multiple transactions and thus stress both resource synthesis and resource allocation. We randomly generate test input packets and inspect the output packets from the simulation. So far, all our sanity checks have passed.

4.7.1 Evaluation Setup and Experiments

We address the following evaluation questions:

Q1: Resource Transformation. How much does CaT's resource transformation help in terms of the resource usage? We select 3 benchmarks [24] extracted from real P4 programs and compare resource usage for pre- and post-transformed programs (§4.7.2).

Q2: Resource Synthesis. How does CaT's resource synthesizer compare to existing ones? We compare CaT with Chipmunk on several dimensions using ALUs drawn from Tofino [68] and Banzai [130], along with controlled experiments on the predecessor packing and preprocessing optimizations (§4.7.3).

Q3: Resource Allocation. How good is the CaT compiler in terms of resource usage? We use Gurobi as the default solver for resource allocation and compare the runtime of the Gurobi

and Z3 solvers. In addition, we compare 2 modes: finding either an optimal or a feasible solution (§4.7.4).

Q4: Retargetable Backend. Can CaT easily perform compilation for different hardware targets? Our synthesis experiments with the Banzai and Tofino ALUs already demonstrate this feature. Additionally, we run the CaT compiler on different *simulated* hardware configurations, compile switch.p4 under varying constraints and report the results (§4.7.4).

Benchmark selection. We use different benchmarks to demonstrate the benefits of each phase of the compiler.

- Resource transformation: 3 benchmarks (ME-1, ME-2, ME-3) extracted from SipHash and UPF (real P4 programs developed by other P4 programmers).
- Resource synthesis: 14 benchmarks together with their semantically equivalent mutations (10 for each benchmark, hence 140 in total) from the Chipmunk paper [50].[3]
- Resource allocation: Same as the benchmarks we use for sanity checking our prototype. We use the full switch.p4 program for experiments that vary hardware resource parameters in the Menshen backend.

Machine configuration We use a 4-socket AMD Opteron 6272 (2.1 GHz) machine with 64 hyper-threads and 256 GB RAM to run all our experiments for both CaT and Chipmunk. Additionally, we note that Chipmunk requires performing a grid search on pipeline geometries (within an upper bound) using multiple such machines in parallel to find an implementation that consumes a small number of pipeline resources. By contrast, CaT does not require multiple machines since CaT's resource synthesis (Algorithm 1) directly tries to minimize pipeline depth without a parallel grid search.

[3]Chipmunk can compile all 14 benchmarks by using Banzai ALUs [130], and 10 of the 14 benchmarks by using Tofino ALUs [50]. For Banzai ALUs, we also show the Domino pipeline usage as reported in the Chipmunk paper [50].

Program	Without CaT transformations			With CaT transformations		
	#gateways	#tables	#stages	#gateways	#tables	#stages
ME-1	15	15	5	15	15	4
ME-2	3	3	3	0	1	1
ME-3	19	12	2	0	3	1

Table 4.4: Resource usage with/without CaT's transformation. GREEN means fewer resource usage.

Program	ALU	CaT					Chipmunk [50]			CaT Speedup	Domino [130]
		Mean Time (s)	Std Dev (s)	Avg #stages			Mean Time (s)	Std Dev (s)	Avg #stages	wrt Chipmunk	Avg #stages
				default	w/o pred	w/o ppa					(from [50])
BLUE (increase) [45]	**Tofino ALU**	19.04	0.43	1	2	1	159.78	59.03	2	8.39 ×	
BLUE (decrease) [45]	**Tofino ALU**	18.72	0.84	1	2	1	142.5	42.5	2	7.61 ×	
Flowlet switching [128]	**Tofino ALU**	19.76	0.69	2		2	962.83	1170.16	2	48.73 ×	
Marple new flow [98]	**Tofino ALU**	6.65	0.52	1		1	5.2	1.71	1	0.78 ×	
Marple TCP NMO [98]	**Tofino ALU**	13.24	0.53	2			6.56	0.36	2	0.50 ×	**N/A**
Sampling [130]	**Tofino ALU**	14.03	0.57	1		1	22.87	10.68	1	1.63 ×	
RCP [142]	**Tofino ALU**	20.19	0.59	1		1	65.13	20.93	1	3.23 ×	
SNAP heavy hitter [19]	**Tofino ALU**	3.58	0.25	1	1	1	26.83	13.63	1	7.49 ×	
DNS TTL change [26]	**Tofino ALU**	20.84	1.97	2	3		36.34	50.55	2	1.74 ×	
CONGA [10]	**Tofino ALU**	10.24	0.43	1		1	3.02	0.17	1	0.29 ×	
BLUE (increase) [45]	**Banzai ALU:** pred raw	40.69	1.41	4	4	4	166.88	36.59	4	4.10 ×	
BLUE (decrease) [45]	**Banzai ALU:** sub	38.83	1.48	4	4	4	1934.82	1611.66	4	49.83 ×	
Flowlet switching [128]	**Banzai ALU:** pred raw	25.37	0.94	3	3	3	185.84	81.41	3	7.33 ×	8.3
Marple new flow [98]	**Banzai ALU:** pred raw	13.79	0.44	2	2	2	12.31	0.18	2	0.89 ×	
Marple TCP NMO [98]	**Banzai ALU:** pred raw	28.12	2.60	3	4		15.3	0.49	3	0.54 ×	
Sampling [130]	**Banzai ALU:** if else	11.52	0.65	2	2	2	33.39	11.09	2	2.90 ×	
RCP [142]	**Banzai ALU:** pred raw	25.08	0.85	2	2	2	31.21	7.55	2	1.24 ×	5.6
SNAP heavy hitter [19]	**Banzai ALU:** pair	3.45	0.23	1	1	1	69.07	19.36	1	20.02 ×	3.3
DNS TTL change [26]	**Banzai ALU:** nested if	32.63	34.91	3	5		211.67	22.65	3	6.49 ×	
CONGA [10]	**Banzai ALU:** pair	10.27	0.55	1	1	1	19.47	8.05	1	1.90 ×	
Stateful firewall [19]	**Banzai ALU:** pred raw	2499.43	4638.58	4	4		6749.89	6349.94	4	2.70 ×	15.5
Learn filter [130]	**Banzai ALU:** raw	31.01	0.73	3	3	3	212.32	4.47	3	6.85 ×	17.5
Spam Detection [19]	**Banzai ALU:** pair	3.51	0.21	1	1	1	59.95	17.75	1	17.08 ×	3.1
STFQ [52]	**Banzai ALU:** nested if	20.99	2.04	3	3	3	22.73	6.94	2	1.08 ×	

Table 4.5: CaT vs. Chipmunk and Domino; Tofino or Banzai ALUs. (pred: Predecessor packing, ppa: Preprocessing, : failed, Std Dev: sample standard deviation). GREEN means faster compilation speed.

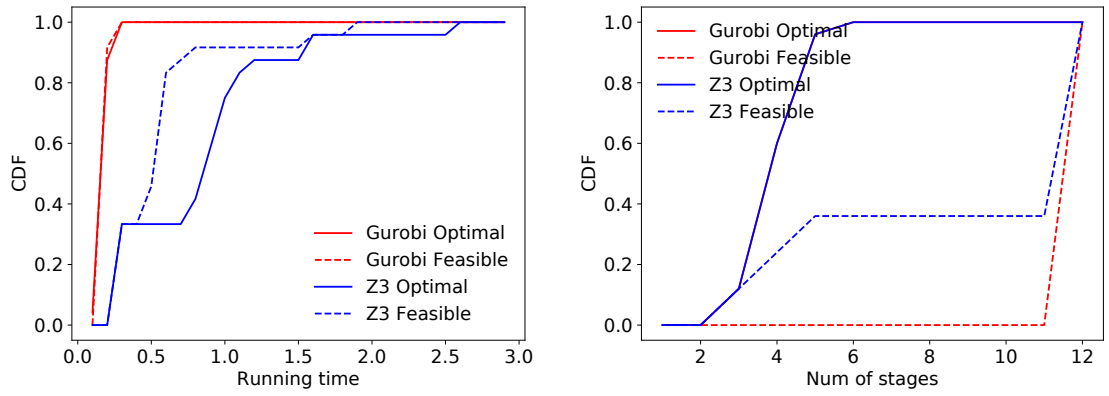

Figure 4.10: Gurobi vs. Z3: Running time, Num of stages.

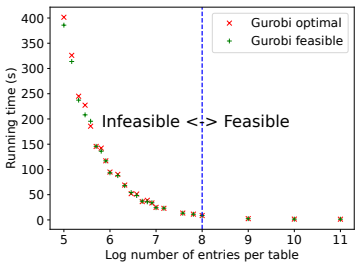

Figure 4.11: Varying # of entries/table.

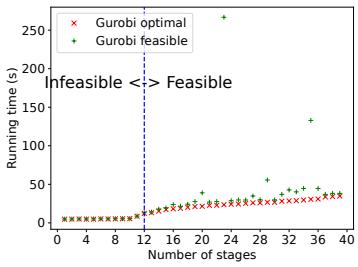

Figure 4.12: Varying # of stages.

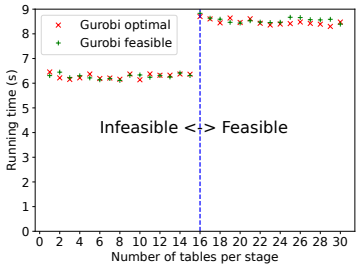

Figure 4.13: Varying # of tables per stage.

4.7.2 Results for Resource Transformation

The resource transformation phase of the CaT compiler performs a best-effort rewrite of if-else statements in the P4 program into match-action tables. Table 4.4 shows the resource usage of compiling benchmarks ME-{1,2,3} to the Tofino architecture with and without the CaT rewrites. As expected, the rewrites help in reducing the number of gateways. Furthermore, they may merge together multiple tables without match entries (i.e., *default* tables), thereby reducing the total number of tables. More importantly, for all benchmarks shown, the rewritten program consumes *fewer pipeline stages* due to either reduced gateway usage (ME-2, ME-3) or the removal of false control flow dependencies (ME-1). CaT does this automatically without the developer engaging in trial-and-error compilation [86].

4.7.3 Results for Resource Synthesis

In all our experiments, the resource synthesis phase consumes the most time, and the SKETCH synthesis queries dominate the overall runtime of CaT. In this section, we focus on evaluating this phase. We compare the CaT and Chipmunk compilers on the SipHash benchmark (cf. Figure 4.2) and on all benchmarks used in the Chipmunk work [50]. For the latter, we target both Tofino ALUs and Banzai ALUs, to evaluate the performance of CaT on different instruction sets and various input programs. This also demonstrates CaT's retargetability via different ALU grammars.

Results for SipHash For the SipHash P4 program, the CaT compiler was successful with the Tofino ALU, and took about 40 hours to complete. In comparison, Chipmunk failed to generate the output even after 150 hours. After investigation, we found two main reasons for the long runtime of CaT: (1) 1 multistage action required 4 pipeline stages – the synthesis query for this action has a large search space and took more than 30 hours in SKETCH. (2) SipHash includes bitvector operations in addition to integer arithmetic. This results in a harder synthesis problem for SKETCH: SipHash uses 32-bit bitvectors, while SKETCH's default for integers is 5 bits.

We plan to explore new ideas for handling deep multistage actions in future work. For handling 32-bit bitvectors more efficiently, we enhanced our basic procedure as follows. We first run the SKETCH synthesis query on a program with a constrained input space, and verify separately whether the generated solution works for the full input space. If it does, then we have found a correct solution; otherwise, we add the generated solution as a counterexample in SKETCH and repeat the procedure. The hope is to quickly generate a solution from the constrained input space that can be proven correct for the whole input space. For the SipHash example, we constrained the input space to use only 16 bits (by setting the higher 16 bits of the input 32-bit bitvectors to 0). CaT separately verified that the generated solution is also correct for unconstrained 32-bit bitvectors. This approach reduced CaT's runtime to under 2 minutes, showing the promise of such an approach.

Results for Chipmunk benchmarks The results are shown in Table 4.5, for Tofino and Banzai ALUs, respectively. We report the runtime of full compilation; for CaT, this includes the resource allocation time, whereas Chipmunk does not perform any resource allocation. We consider 10 semantically equivalent mutations of each of the benchmarks, which are identical to those in Chipmunk [50]. We report the mean and sample standard deviation of compilation time across all mutations.[4] These experiments evaluate if CaT can effectively handle semantically equivalent

[4]The runtimes in Table 4.5 are similar to, but slightly different from that in Table 2 of the Chipmunk paper [50]. The differences arise due to Chipmunk's use of SKETCH's parallel mode, which introduces non-determinism due to thread interleaving.

programs.

We report the resource consumption of the generated code produced by CaT, in terms of the total number of pipeline stages required on average across the mutations ("#stages default" column). Stages are the most scarce resource in programmable switches (e.g., 12 for Tofino). For evaluating the effectiveness of our predecessor packing optimization (pred) and the preprocessing analyses (ppa) (§4.5.3), we also report the number of stages without these optimizations in columns "#stages w/o pred" and "#stages w/o ppa." Gray-ed entries indicate a difference from the default setting.

Our results show that CaT is able to compile *all* programs successfully compiled by Chipmunk, with almost all compiled results having a matching number of pipeline stages. Furthermore, *CaT is often much faster and more stable (in running time) than Chipmunk*. Specifically, for the Tofino ALUs (top section of Table 4.5), CaT finishes compilation within a few seconds, 2.75x faster on average (geometric mean) than Chipmunk. The max speedup is 48x for *flowlet switching*, a minutes-to-seconds improvement (≈16 minutes in Chipmunk vs. 20 seconds in CaT). In *BLUE (increase)* and *BLUE (decrease)*, CaT generates a solution with fewer stages than Chipmunk. In all other benchmarks the number of stages is the same. For the BLUE benchmarks, since the Tofino stateful ALU contains two registers, CaT's optimizations enabled it to pack a successive pair of stateful updates into a single stateful ALU (§4.5.3). In comparison, Chipmunk mapped the two stateful updates to two ALUs in two stages. This shows that CaT's approach can find additional opportunities for fully utilizing the functionality of available hardware resources. Predecessor packing is also effective in 9 of 10 benchmarks, enabling compilation to succeed or reducing the number of stages; preprocessing is also useful in 2 benchmarks.

For the Banzai ALUs (lower section of Table 4.5), we additionally report results on stages output by the Domino compiler [130] (which only handled Banzai ALUs), with the average number of stages across different program mutations shown in the last column (as reported in [50]). Note first that CaT takes no more than 1 minute on most successful benchmarks. Although it takes 40

minutes for *stateful firewall*, Chipmunk is much slower, requiring more than 1.5 hours. CaT provides 3.94x speedup on average (geometric mean) and 49x maximum, with respect to Chipmunk. CaT is slower only on *Marple new flow* and *Marple TCP NMO*, but finishes both within 30 seconds. Note that Chipmunk must use multiple machines in parallel for synthesis, while CaT only uses one machine for synthesis. In terms of number of stages, CaT generates code with the same number of stages as Chipmunk for all benchmarks except the *STFQ* example (3 in CaT vs. 2 in Chipmunk). Upon investigation, we find that this is due to separation between queries for stateful and stateless nodes in our synthesis procedure. Although our predecessor packing optimization can often mitigate this negative effect, we plan to improve it further in future work. Still, both predecessor packing and preprocessing optimizations are effective in some benchmarks here as well. Finally, Domino either fails to compile (8 of 14 examples), or uses many more stages (other 6 examples). *Overall, CaT generates high-quality code comparable to Chipmunk, but in much less time and with fewer compute resources.*

Results for controlled experiments. We selectively turned on 2 optimizations: (1) Predecessor packing, (2) Preprocessing analyses (constant folding, algebraic simplification, dead code elimination). According to the results in Table 4.5, for Banzai ALUs, without predecessor packing, our compiler uses additional stages in two examples (*Marple TCP NMO*, and *DNS TTL change*), showing that predecessor packing can reduce the number of pipeline stages; for the Tofino ALUs, predecessor packing was even more beneficial: disabling predecessor packing resulted in compilation errors for 6 examples (*flowlets, Marple new flow, Marple TCP NMO, Sampling, RCP*, and *CONGA*). The reason is that the Tofino ALU supports very limited stateless computations and cannot handle relational or conditional expressions. Packing such expressions into adjacent stateful ALUs was essential for compilation to succeed. For Banzai ALUs, without preprocessing analyses, 3 of the examples could not be compiled. The runtime of preprocessing is less than 0.1 sec in all examples. *Overall, CaT's optimizations allow compilation to succeed where it would fail otherwise and reduce the number of pipeline stages.*

4.7.4 Results for Resource Allocation

We experiment with two solvers (Gurobi and Z3) and two modes (optimal and feasible) on our benchmark examples. The results are in Figure 4.10 with more detailed data. The results show that for checking feasibility, Gurobi returns suboptimal solutions that use all the pipeline stages, while Z3 finds feasible solutions that are better than Gurobi's but takes marginally more time. However, Gurobi finds an optimal solution almost as quickly as a feasible solution. For these benchmarks, Gurobi is faster than Z3. Thus, Gurobi with optimization is a good default.

In additional experiments, we study the resource allocation time of switch.p4 as a function of the parameters of the Menshen backend target. We vary the maximum number of entries per table, number of stages, and number of tables per stage, and plot the runtime of Gurobi in both optimal and feasible mode in Figures 4.11, 4.12, 4.13. A vertical line indicates the transition from infeasibility to feasibility for the constraint solver. Across a variety of hardware configurations, we find that the runtime of both modes are quite similar. Figure 4.11 shows that runtime increases as the maximum number of entries decreases because of an increase in the number of partitions of a table as the maximum number of entries decreases. Figure 4.12 shows that runtime increases as the number of stages increases because of the increase in the number of indicator variables tracking which stage a table belongs to. In Figure 4.13, the number of Gurobi variables is constant as we vary the number of tables per stage; The runtime is similar for optimal and feasible modes, but varies significantly depending on whether there is a solution.

4.8 Summary

We introduce a new decomposition of the compilation problem for packet pipelines into 3 phases: resource transformation, resource synthesis, and resource allocation, where solver engines (e.g., ILP, SMT, program synthesis) are employed extensively within these phases. We pro-

totype CaT, a compiler for P4 programs based on this decomposition. CaT can handle more programs, reduce pipeline resource usage, compile faster, and requires fewer compute resources than existing compilers. We hope our results encourage compiler engineers for such pipelines to adopt similar ideas.

5 Cross-Platform Transpilation of Packet-Processing Programs using Program Synthesis

The proliferation of programmable network devices offers a wide range of device options for developers of packet processing programs. However, there are several differences in programming language usage, hardware resource constraints, and hardware architecture across these devices. Programmers must understand multiple programming languages and hardware designs to write programs for various devices.

We propose an alternative: leveraging program synthesis to build a transpiler, Polyglotter, that outputs programs for target hardware devices from input programs written for source hardware devices. This can reduce the efforts required to write algorithms across platforms. Our evaluation results show that, compared to traditional program rewriting methods, Polyglotter can quickly produce correct results with efficient use of hardware resources for the parser part of a packet-processing program. We want to extend it to realize transpilation for the whole network program with the pipeline operation included.

5.1 Introduction

Programmable network devices are becoming increasingly popular because of their flexibility in supporting customized network functions and their ability to handle packet processing workloads with high throughput. Devices including Barefoot Tofino [114], Broadcom Trident 4 [148], the Intel IPU [69], and the Pensando SmartNIC [109] share similar architecture (e.g., RMT pipeline [30]) but have varying hardware resource constraints. Although the emergence of these hardware devices provides more choices, the diversity of their features increases programmers' difficulty in writing code for each specific device.

To alleviate the difficulty of writing programs for multiple hardware devices, we propose automatically generating these programs using a program synthesis-based transpiler called Polyglotter. Given as input a program written for a source device, Polyglotter will take into consideration the hardware constraints of the target device before generating programs that can run on the target device.

Program synthesis tools use an exhaustive search algorithm to find an output program that is semantically equivalent to the input program. Compared with performing transpilation based on numerous rewriting rules, there are several benefits to incorporating program synthesis into the transpiler. First, it is impractical for humans to exhaustively list all rewriting rules without ignoring corner cases. This is why we believe that traditional pattern-matching compilers, consisting of many program rewrite rules, need to be repeatedly updated by adding new rules. Second, even if we could list all possible rewriting rules, the program synthesis-based approach can output more resource-efficient results as it can search through all feasible solutions within the hardware resource constraints. Manually-developed rules may guarantee semantic equivalence but cannot ensure that the outcome is ideal in resource usage. Exceeding available resources, such as pipeline stages in a programmable switch, can lead to transpilation failures.

The workflow of our transpiler is represented in Figure 5.1. To design our transpiler, we use

hardware configuration files to help the transpiler interpret the semantics of the input program and generate programs that follow the target hardware device's resource constraints. This requires the transpiler developers to be familiar with the source and target hardware constraints (§5.2); and the programming language semantics for the input and output programs (§5.2). They need to ensure that output programs do not exceed the target devices' resource limit and are semantically equivalent to input programs. All of these should be encoded into our transpiler.

However, building the transpiler using program synthesis requires overcoming a key challenge: the long running time due to the large search space of all implementation candidates. Hence, our design divides the whole synthesis problem into several smaller steps, each of which only solves a subproblem (§5.3.3.4), either state transition or packet field extraction, of the whole transpilation, leading to a faster transpilation speed.

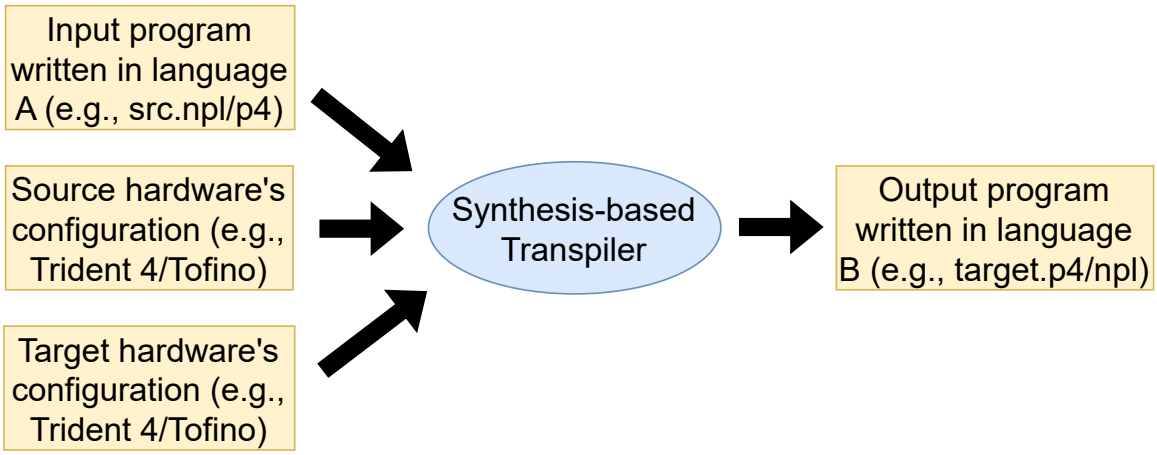

Figure 5.1: Workflow of the transpiler design where language A and B can be the same or different.

We test this idea over P4 programs and NPL programs that exercise the packet parser on the Tofino switch [114] and Trident 4 switch [148]. The preliminary results (§5.4) show that Polyglotter can quickly generate correct target programs that are semantically equivalent to source programs. Compared with transpilation using several rewrite rules, our output is more efficient in terms of resource usage. We also outline several future directions to extend our approach to

entire packet processing programs including the parser, pipeline, and stateful packet processing functions across devices.

5.2 Where is Transpilation Useful?

A transpiler should tackle the differences between the source/target programming language designs and source/target hardware constraints. Below, we list several examples of language and hardware differences that necessitate transpilation, with source and target language snippets. When these examples occur concurrently in one program, there could be a combinatorial explosion for the total number of semantically equivalent output options, each with different usage for different types of hardware resources.

5.2.1 Wide state transition key

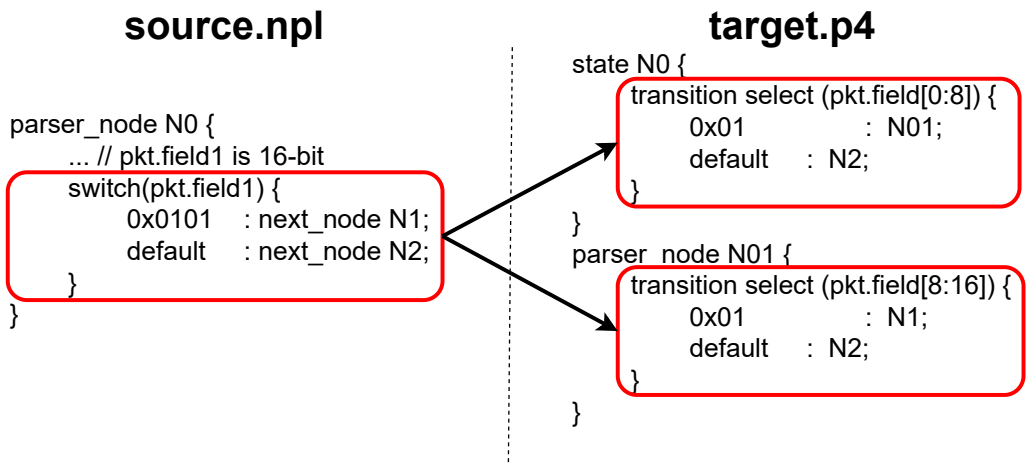

Figure 5.2: Transpiling a parser node with wide match key (16 bits) into multiple parser nodes with narrower match key (8 bits).

A hardware packet parser identifies headers and extracts packet fields for subsequent processing in the ingress/egress switch pipeline. A parser is commonly modeled as a finite state machine (FSM). Each state is a *parser node*. Each parser node can take a packet field as its state transition

key and transit to subsequent states based on the key's value. However, hardware limits the bit width of a state transition key and this limit varies across platforms. We may have to convert one parser node on a particular hardware into multiple parser nodes on a different hardware.

As a concrete example shown in Figure 5.2, *N0* from the input program on the LHS has a transition key `pkt.field1` of size `16` with 2 switch cases. The language of the output program has a limit on the transition key to be of size ≤ 8. As a result, the transpiler needs to use 2 parser nodes, each of which checks *8 bits* of the transition key. One solution is presented on the RHS of Figure 5.2. In general, when the size of the transition key is larger than the transition key width limit of the target hardware, multiple parser nodes are required in the transpiled program to express the same semantics.

5.2.2 Table operations in parser

In a parser, the allowed operations might be different across languages and hardware devices. In NPL programs, the language allows `parse break` and `parse continue` to allow interleaving parser and logical table operations. Specifically, the Trident switch hardware jumps out of the parser after `parse break`, does a series of logical table operations, and jumps back to the parser after `parse continue`. However, such interleaving is not allowed in P4 programs. So the transpiler should express the same behavior in a different manner that is supported by the P4 programming language.

The logical table in NPL performs match action operations based on rules from the control plane. If we put one logical table between 2 parser nodes, the variables updated in the logical table might influence the behavior of the subsequent parser node. We observe that, depending on the rules programmed by the control plane, a P4 `value_set` may be used to mimic the same semantics of an NPL table that influences the behavior of a downstream parser node. A P4 `value_set` is a data structure that may be used as part of a parser transition to check whether or not the current value of a packet field belongs to a set.

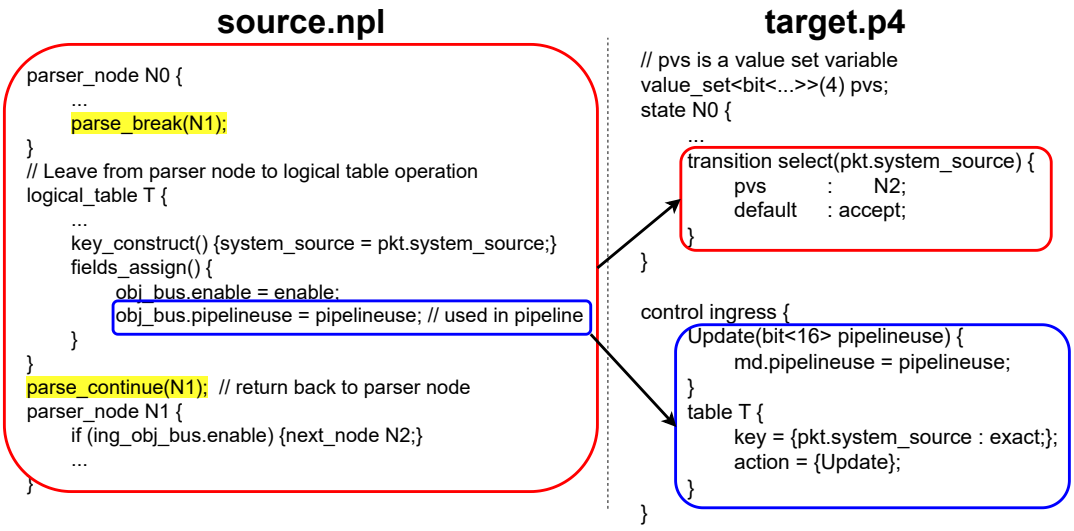

Figure 5.3: Transpiling parse break/continue into the a value set data structure and table operation in a pipeline.

The NPL program in Figure 5.3 has one logical table between 2 parser nodes (parser N0 and parser N1). This table updates a bus field (`obj_bus.enable`) that is used in parser node N1. Even though P4 does not support table operations within its parser, it can implement the same semantic by using `value_set`. In this example, checking the updated value of the bus field is equivalent to checking the execution status of the logical table. Thus, we could know the updated value by checking whether the key of the logical table is in the match set or not using `value_set`.

Additionally, logical tables within the parser in languages such as NPL might update variables that are used in the pipeline. In Figure 5.3, `obj_bus.pipelineuse` is updated in one logical table of the parser and used in the pipeline later. Given P4 language cannot do temporary variable updates in the parser, the generated transpiled program adds one extra P4 table at the beginning of the pipeline to guarantee that the rest of the pipeline can witness the latest value of `obj_bus.pipelineuse`. There could be other ways to generate semantically-equivalent transpiled programs as well.

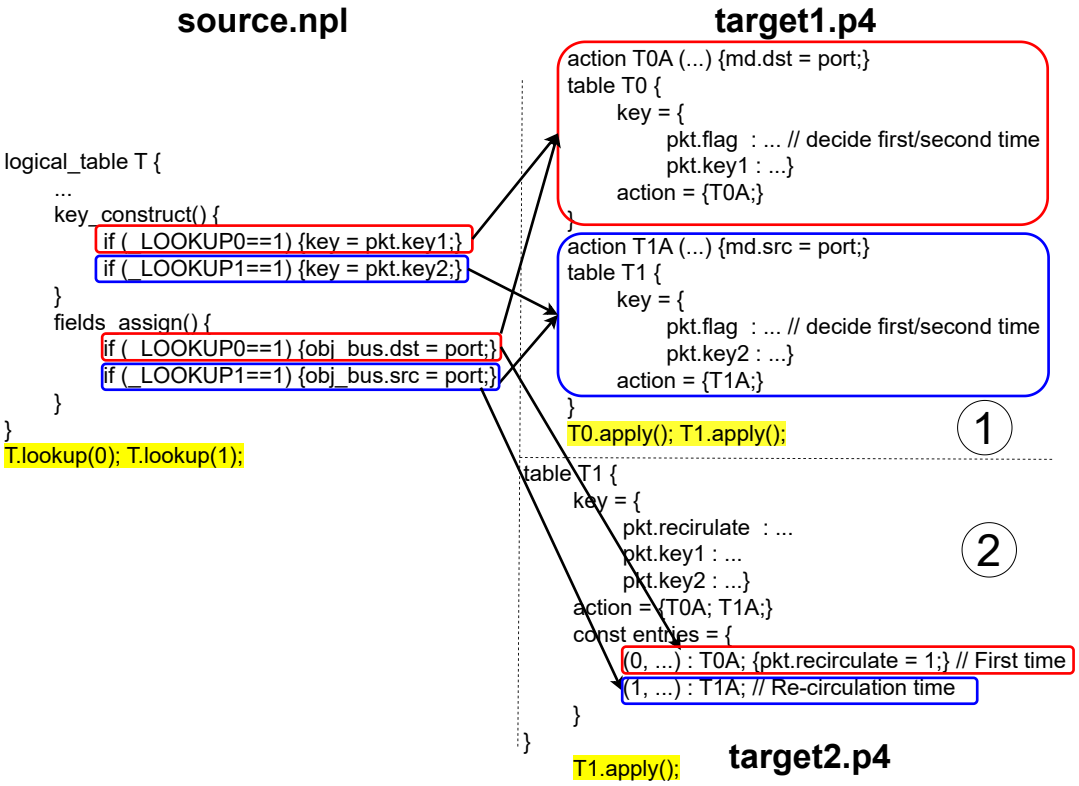

Figure 5.4: LHS shows an NPL program with 2 lookups for 1 logical table; the RHS shows 2 alternatives for the transpilation results in P4.

5.2.3 Multiple lookups per table

We use packet headers and temporary variables to store intermediate information during the parsing process. We use the table data structure to update packet headers or temporary variables based on match-action rules. But there are differences in the # lookups per table across languages (e.g., NPL vs P4).

The program on the LHS of Figure 5.4 shows one logical table in NPL which can support 2 lookups. Allowing more than one lookup per table can let programmers write more complex functions into one table without using multiple tables. More benefits of this feature in the NPL language can be seen from the NPL spec [99]. However, P4 can only support one lookup for each table. This restriction forces the transpiler to find non-trivial ways to express the same

functionality in P4.

We show two methods to implement the multiple lookup functionality in P4. The method labeled ① in Figure 5.4 defines 2 tables, each containing a copy of the match-action rules in the NPL table. Each table maps to one lookup in the NPL's logical table. We need to add one extra boolean match key `flag` in each P4 table. If its value is 0, it maps to the first lookup from NPL; otherwise, its value is 1, meaning it is the second lookup from NPL.

It is also possible to use packet re-circulation to transpile multiple lookups. ② of Figure 5.4 realizes such an implementation using only 1 table by adding an extra match key `pkt.recirculate`. The benefit of re-circulation is that it can store useful information in a first pass and continue processing packets in a second pass. But this may reduce the throughput. The choice of which transpilation method to use depends on the programmers' objectives.

5.2.4 Initialization for temporary variables

The initialization approaches for temporary variables are different in the NPL and P4 languages. Specifically, the NPL initializes all temporary variables (called bus fields in the NPL program) through one initialization function before the parser while P4 does so by extracting bits from the input bit stream at the beginning of the parser. This difference leads to a potential failure in transpilation.

Figure 5.5 gives a concrete example. `source_fail.npl` initializes `ing_obj_bus.field1` and the initial value decides transition logic in the parser node `start`. We cannot find an equivalent expression in P4 due to its language design constraints. However, `source_success.npl` initializes `ing_obj_bus.field1` but its value is later updated by one extracted packet field. We can express this semantic through packet extraction shown in `target.p4`.

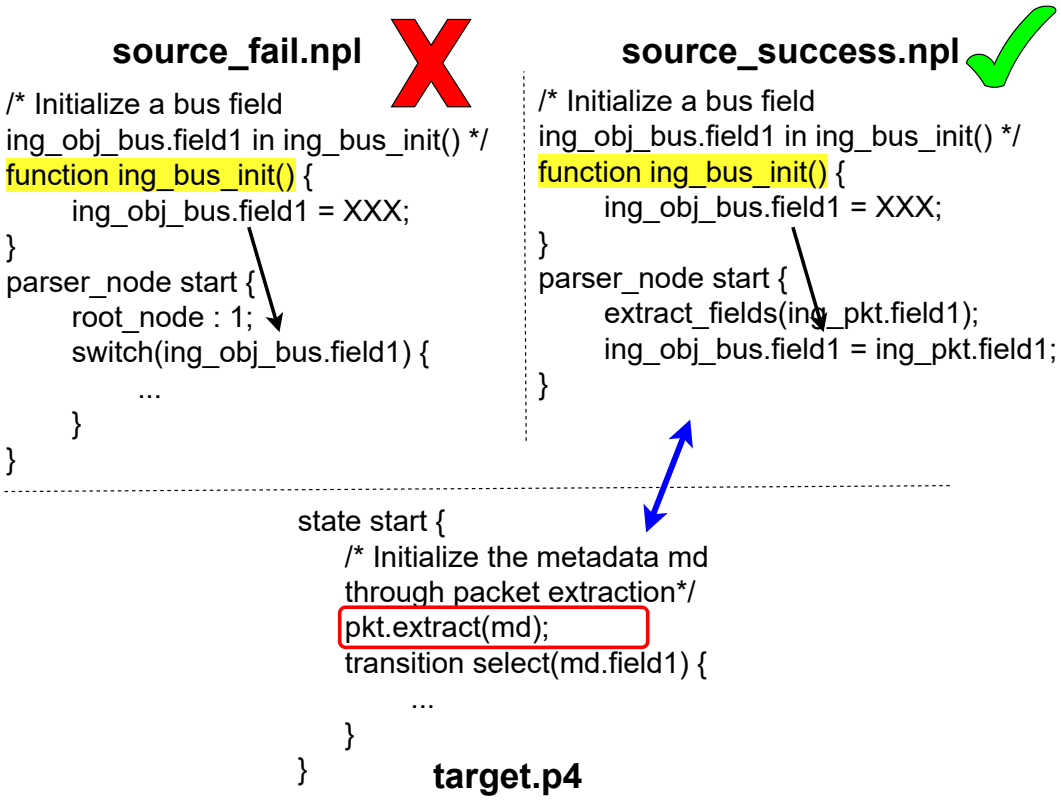

Figure 5.5: Different ways to initialize temporary vars.

5.3 Work: Automated Parser Transpilation

We build Polyglotter, a cross-platform transpiler for the parser portion of NPL and P4. This requires us to handle transpilation examples such as those mentioned at §5.2.1 and §5.2.2. All other cases described in §5.2 are beyond the parser and left for Polyglotter's future development. It consists of 3 main steps (Figure 5.7). In step 1, Polyglotter generates the corresponding intermediate representation (IR) from the input program; step 2 turns the IR format into a semantically equivalent version that follows the target hardware's constraints using program synthesis; in step 3, Polyglotter lifts the synthesized IR format back to the target programming language.

Literals:	c	::=	NUMBER
Extraction status bit vector:	bv	::=	BIT VECTOR
Bit vector:	I	::=	INPUT BIT VECTOR
Value sets:	S	::=	$S_1 \mid S_2 ... \mid S_{k1}$
Packet fields:	pkt	::=	$f_1 \mid f_2 ... \mid f_{k2}$
Temp variables:	tmp	::=	$t_1 \mid t_2 ... \mid t_{k3}$
Operations:	op	::=	"==" | "in" | "="
Expressions:	e	::=	$c \mid pkt \mid tmp \mid bv[c] \mid S \mid I[c:c] \mid$ e op e
Predicates:	$pred$	::=	e (&& e)* $\mid TRUE$
Statements:	s	::=	$e \mid s\ ;\ s \mid \epsilon \mid$ `if` $(pred)\{s\} \mid$
			`if`$(pred)\{s\}$`(elif(pred){s})`* `else`$\{s\}$
Packet fields definition:	$pktDef$	::=	bit<c> $pkt;\ pktDef \mid \epsilon$
Temp variables definition:	$tmpDef$	::=	bit<c> $tmp = c;\ tmpDef \mid \epsilon$
Bit vec definition:	$bvecDef$	::=	bit<c> $bv = \{0, 0, ..., 0\};\ \mid \epsilon$
$p \in$ program		::=	$pktDef;\ tmpDef;\ bvecDef;\ s$

Figure 5.6: The BNF of IR for parser.

5.3.1 IR design for parser behavior

Designing a new IR format can simplify complex data structures (e.g., logical table in NPL and value set in P4) and help extend the transpilation process across more hardware platforms and programming languages. Verifying the semantic equivalence at the IR level is much simpler and more generic.

The parser *determines* packet headers from an input bit stream through a sequence of parser nodes. Therefore, the IR design should be capable of reflecting the parser's behavior (e.g., state transition and packet header extraction). Inspired by [124], Figure 5.6 shows the IR grammar. The general workflow of a parser starts from variable declaration for packet fields, temporary variables, and extraction status bit vectors. Then, the whole parser will be represented by a nested if-then-else (ITE) block.

Parser node transition. We use a sequence of ITE blocks to model the parser behavior. In the nested ITE statement, the predicate consists of at least one clause and each clause checks whether the value of one variable is equal to a constant or belongs to one set. We include “True" as a possible predicate for completeness. All these predicates are concatenated by “&&". The “||" logic can be represented by using the *elif* statement it is not in the predicate IR. If the predicate is satisfied, it will enter into the next level ITE block, meaning the transition to the next parser node; the *else* keyword can be considered as the *default* statement in the parser.

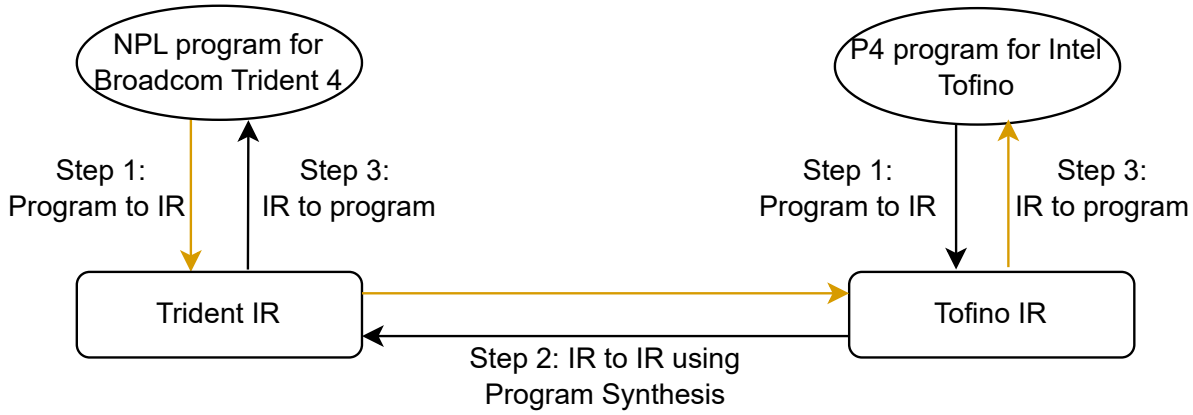

Figure 5.7: 3 steps in Polyglotter’s transpilation.

Packet field extraction. There are 3 types of operators (e.g., ==, in, and =) in the IR design, and the operator “=" is used to define the packet field extraction behavior. To be specific, *bv* is a set of indicator variables, each of which represents whether a corresponding packet field is extracted or not. Setting its value to 1 means this packet field should be extracted in the parser node. Then, we update a packet field’s value by setting it to one slice of an input bit stream using *I[c:c]*. Our current IR design is expressive to express parsers for both Tofino and Trident 4 programmable switches.

5.3.2 Step 1: Generating low-level IR

In this IR generation step, developers need to make sure that the generated IR is semantically equivalent to the input program and complies with hardware capabilities. Polyglotter analyzes the input program's semantics and outputs the IR format. This process involves predicate generation and packet extraction generation.

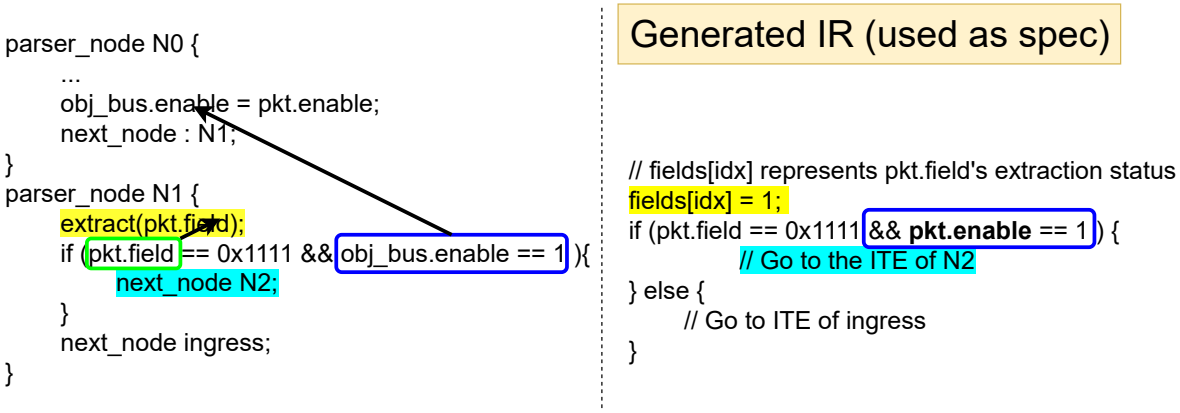

Figure 5.8: Predicate generation.

Predicate generation. In our IR design (Figure 5.6), one predicate might include several clauses and each clause is to compare a variable (e.g., packet field or temporary variable) against either a constant or a set. In hardware such as Tofino, it does not support temporary variable modification in the parser so we should treat packet fields and bus fields differently. Specifically, if the variable is a packet field, we should leave it as it is. However, when the variable is a temporary variable, Polyglotter does a backward dataflow analysis to find all possible packet bits that influence this temporary variable, and then replaces the predicate by one or more conditions over those packet bits.

As a concrete example, the predicate in Figure 5.8 has 3 clauses "`pkt.field==0x1111 && obj_bus.enable==1`". The predicate replacement pass leaves the packet field (e.g., `pkt.field`) as it is. However, it has to determine the packet bits that affect the value of bus fields. Given the temporary variable `obj_bus.enable` is affected by `pkt.enable`, the clause `obj_bus.enable==1`

is replaced by `pkt.enable==1`.

Packet extraction generation. Each parser node might extract a range from the input bit stream and set values to several packet fields. Our IR design regards the whole parser as a nested ITE block. Polygotter uses one *bit vector* to represent each packet field's extraction status. In the output IR, we represent the packet extraction behavior by setting the corresponding indicator variables to 1.

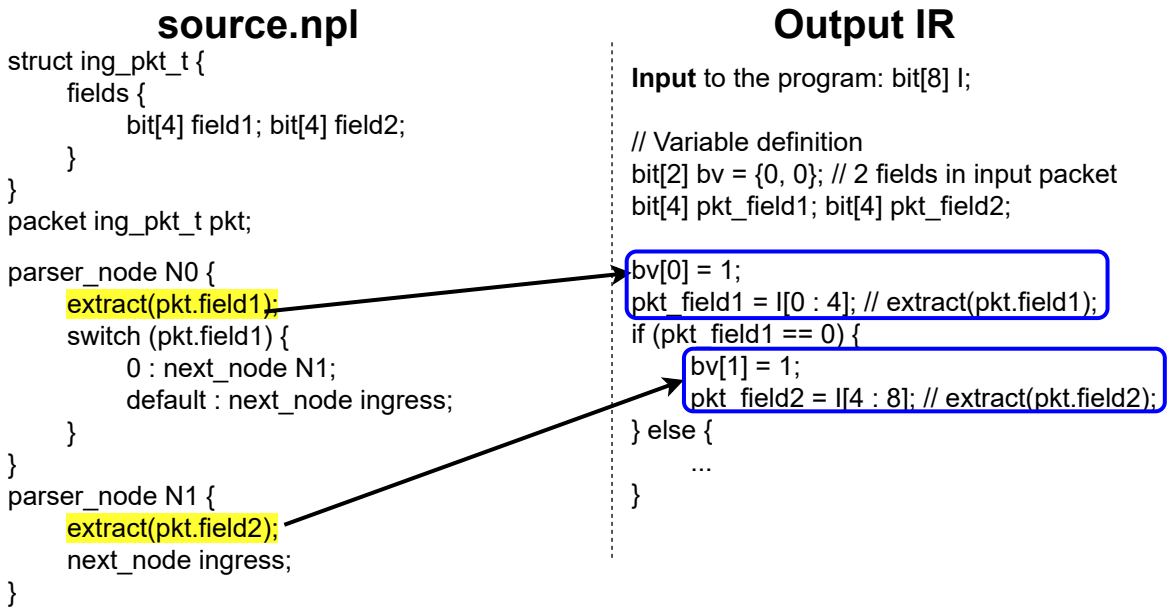

Figure 5.9: The generated IR for one given input NPL program.

Output IR from input program. After predicate generation and packet extraction generation, we can output the IR format. Figure 5.9 provides one concrete example of the IR generation process for an NPL program. Specifically, in the output IR, we use a bit vector variable, *bv*, and some other bit vector variables, *field1* and *field2*, to record the extraction status of each packet field. There are 2 fields in the input program so the size of the bit vector is 2 as well. All other bit vectors are used to store the extracted packet fields' value. The whole parser is presented by a large "nested" ITE statement, where each if predicate is one state transition logic, determining the subsequent packet extraction behavior of the parser.

5.3.3 Step 2: Generating IR for target device

Polyglotter encodes hardware constraints and generates the target IR format through program synthesis. The output program not only guarantees the semantic equivalence, but also obeys the resource constraints. We divide the IR to IR transformation into 2 parts: **synthesis for predicates** and **synthesis for packet extraction**.

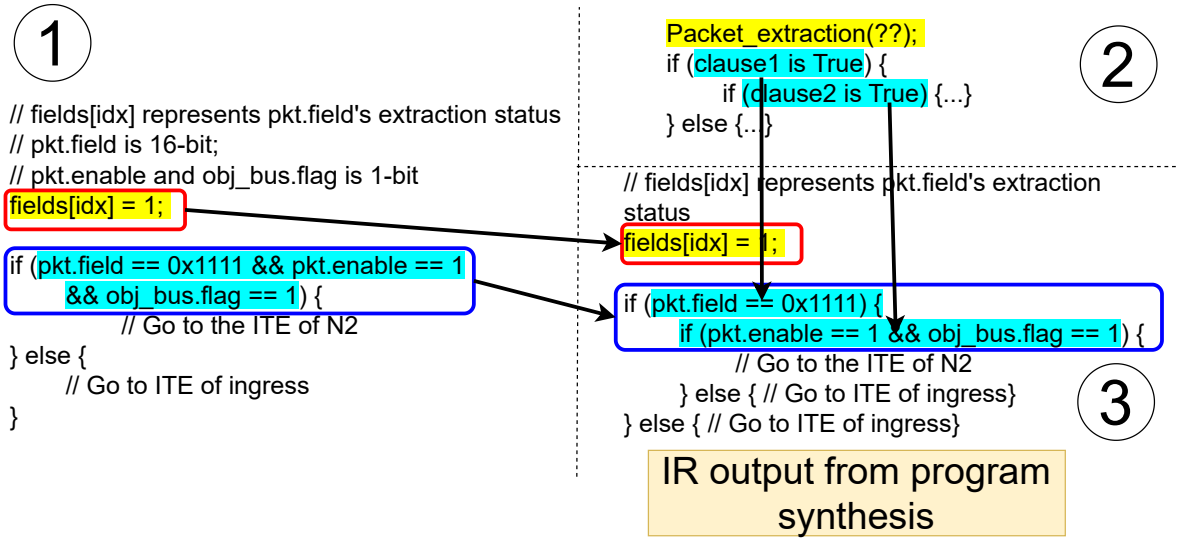

Figure 5.10: Synthesis-based IR generation for target. ① is the input program; ② is the output after the synthesis for predicates; ③ is the output after the synthesis for packet extraction.

5.3.3.1 Synthesis for predicates

Polyglotter splits the predicate synthesis into multiple sub-problems, each of which takes all predicates within *one parser node* as the specification. Hardware resource constraints such as the size limit of a state transition key will be encoded. As an example, if the target device limits the transition key to be at most 16 bits, we cannot fit the if-condition (the transition key is 18 bits) in ① of Figure 5.10 into 1 parser node and need to use 2 in ② of Figure 5.10. In addition, Polyglotter captures common conditions in all if-predicates and places them before other conditions in the generated IR. Its benefits will be discussed in §5.3.3.4.

5.3.3.2 Synthesis for packet extraction

The next step is to decide where in the process of parsing we extract specific packet fields. In this case, the specification is the generated IR after predicate synthesis where we could replace all conditions with one boolean variable; while the implementation is a partial program where we put one packet extraction function in each if-else body. The function will decide what new fields to extract in this node. In ③ of Figure 5.10, the transpiler represents the packet field extraction behavior by setting the indicator variable `fields[idx]` to 1. Polyglotter ensures that each packet field is extracted at most once in the parser.

5.3.3.3 Transpilation Algorithm

We show the detailed algorithm to generate IR for the target devices in Figure 5.11. There are 3 major steps. The first step generates an if-then-else (ITE) representation for one parser node; then the whole parser will be in the format of a nested ITE skeleton after replacing each parser node with its ITE representation in step 2; finally, step 3 fills the packet extraction behavior into each body of the ITE skeleton. The program synthesis engine provides the semantic correctness guarantee for the transpiled program.

Input:
1. Parser program $P(N, A, E)$, with several nodes $N =$ Parser Nodes, state transition rules $A =$ Transition Arrows, and packet header extraction functions $E =$ Packet Field Extraction functions;
2. The total size limit for state transition keys in one parser node $keyWidth$;

Output: Synthesized code for the whole parser program that follows the target device's hardware constraints.

1 // Step 1: Generate IR that follows target devices' constraints for each parser node.
2 $i \leftarrow 0$
3 $nodeToITE \leftarrow \{\}$
4 **for** $n \in N$ **do**
5 | `ITE Skeleton i` $\leftarrow$ `querySketchNode` $(n, ITE_i(shared, unique), keyWidth)$
| $nodeToITE[n] \leftarrow$ `ITE Skeleton i`
6 **end**
7 // Step 2: Output a big ITE skeleton containing all parser nodes within the parser
8 **for** $n \in N$ **do**
9 | // Update all nodes in P
10 | $n \leftarrow nodeToITE[n]$
11 **end**
12 $ITESkeleton \leftarrow P(N_{new}, A, E)$
13 // Step 3: Generate packet extraction for each body of ITE
14 $OutIR \leftarrow$ `querySketchExtract` (P, ITESkeleton)

Figure 5.11: Algorithm to generate IR for the target devices.

5.3.3.4 Why do we use the program synthesis-based approach?

Synthesis for ITE predicates. Polyglotter generates the implementation of all ITE predicates within a parser node at one time. Usually, the state transition logic within one parser node is not that complex so the synthesis engine can output the result quickly. Besides, compared with a manual approach to creating the result, using synthesis tools can reduce resource usage (e.g., # transitions and # nodes). For example, in Figure 5.12 the input program does a parser state

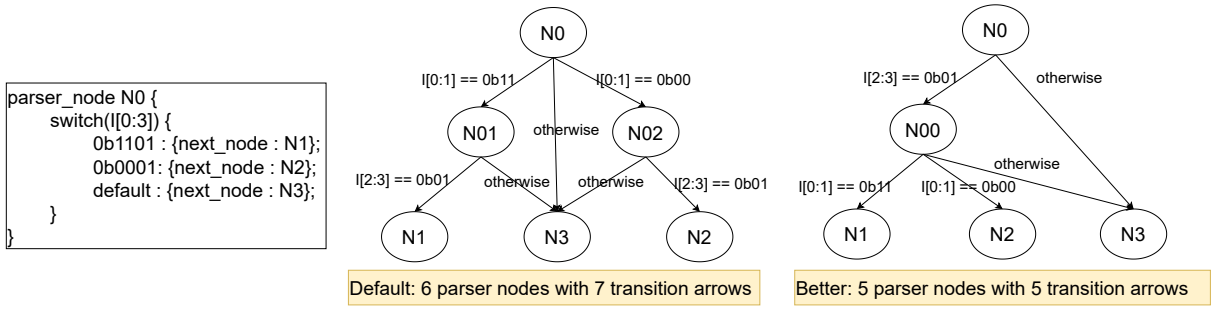

Figure 5.12: Two semantically equivalent transpilation results with the limit of transition key size to be 2 bits. Each result uses different resources.

transition depending on one 4-bit variable. The target hardware can match at most 2 bits in one parser node. One way is to match the first 2 bits and then the next 2 bits in sequence before deciding the next transition parser node. This example is illustrated as in the first tree-based finite state machine which uses 6 parser nodes and 7 transition arrows. However, the right figure shows an alternative, which only needs 5 parser nodes and 5 transition arrows by matching the last 2 bits before matching the first 2 bits. In our synthesis procedure, we let the synthesis engine find common conditions among all predicates and the transpiler will prioritize putting these common conditions before others.

Synthesis for packet extraction. We let the synthesis solver determine the concrete extraction behavior in each generated parser node. Given there are several paths from the first parser node to the last one, it is both time-consuming and error-prone for humans to manually figure out a

solution. However, a synthesis solver can generate the correct result quickly.

Benefits of decoupling predicate synthesis and packet extraction synthesis. Compared to generating the whole parser from one synthesis problem, we need to do implementation configurations exploration. Each configuration is determined by different hyper-parameters (e.g., the number of state transition rules per parser node, the value of each constant value, and the total number of parser nodes). Instead, Polyglotter generates the IR format for state transition predicate and packet extraction in sequence. Synthesizing all ITE predicates can give us the ITE skeleton and the subsequent synthesis is to fill packet extraction behaviors into each ITE's body. Therefore, such decomposition not only makes the transpilation faster but also avoids the skeleton exploration work.

5.3.3.5 Why do we use different granularity for predicate and extraction synthesis?

We synthesize all predicates within *a parser node* because common conditions may exist among all these predicates. There are other alternatives. For instance, we could synthesize predicate by predicate, but this may lose the benefits of finding common conditions from these predicates; if we synthesize all paths within the whole parser together, there may be no conditions shared by them. Synthesizing all packet extraction operations for *the whole parser* is a good balance. First of all, the search space is proportional to # packet fields and # parser nodes, which is in a reasonable size; additionally, this choice can leverage the advantage that some packet extraction behavior can be shared across paths instead of appearing multiple times.

5.3.4 Step 3: Lifting to switch program

The last step is to turn the IR back to a complete program written in the target language that can run over the target hardware. This step can be considered as the reversed process for step 1. The concrete implementation is just a straightforward rewriting process. For instance, in our IR design, we use indicator variables to determine the extraction status of a packet bit but in a

high-level program, we should replace it with corresponding keywords that follow the language syntax. Currently, we manually transform the synthesized output IR to the high-level programs in the target language.

5.4 Evaluation

We measure Polyglotter in 2 main parts. **Correctness**: can it correctly generate the output program within a reasonable time period? **Efficiency**: how many resources does the output program consume?

Setup. We generate benchmarks by extracting portions of one NPL program used in production. All these benchmarks reflect some features mentioned in (§5.2) because we want to check whether Polyglotter can successfully realize the transpilation for programs with these features. The syntax of the output P4 program is checked by the commercial Tofino compiler and the semantics are verified manually. Currently, Polyglotter focuses on the parser portion and realize the transpilation from a NPL program for the Trident 4 switch to a P4 program for the Tofino switch. We plan to extend it into the pipeline portion and realize multi-directional transpilation across languages in the future.

Results. Preliminary results are in Table 5.1. The rule-based transpilation implements a series of rewriting rules to get the output program, including dividing a condition with a big transition key into several conditions with small transition keys in order. All these rules might generate locally optimal results but the combination of several rules can be far from the globally optimal outcome. In fact, Polyglotter can successfully generate correct output for 4 representative benchmarks that fit the target hardware's constraints within seconds. In terms of the resource usage, Polyglotter can output results that use fewer number of # nodes, # transition, and the depth of the parser. In addition, it is useful to decompose the whole synthesis problem into 2 subproblems (e.g., predicate and packet extraction) to speed up the transpilation procedure.

Program	Rule-based transpilation			Polyglotter		
	# nodes	# transition	depth	# nodes	# transition	depth
Wide Key	6	7	3	5	5	3
Parser break/continue	3	3	3	1	0	1
Temporary variable initialization	1	0	1	1	0	1
Move update to pipeline	2	2	2	2	1	2

Table 5.1: Resource usage comparison. GREEN means better resource usage.

5.5 Related Work

Programming languages for programmable network devices. P4 [104] and NPL [99] are widely used programming languages for engineers to develop programs for high-speed network devices. Many domain-specific languages (DSLs) have emerged in academia to remedy some of the shortcomings of P4 and NPL. Examples include Lyra [46], microP4 [137], Domino [130], Lucid [135], FlightPlan [140], P4All [60] and NetKAT [16]. Different languages serve different design purposes, such as offering convenient data structures [60] and incorporating the target hardware's specific features [46] [50]. Our work, with a focus on cross-platform transpilation of existing DSLs, is complementary to the design work of new DSLs. This paper aims to unify features of different languages and hardware devices by building a cross-platform transpiler. This could avoid forcing programmers to understand several language-specific and hardware-specific features.

Program synthesis for compilers. Program synthesis [54] selects one program from a space of programs such that this program is semantically equivalent to the input specification. This technique was used in compiler design (e.g., Chipmunk [50] and CaT [48]) for programmable network devices. However, their focus is to compile a high-level packet processing program into low-level representation without considering the parser portion of the program. In addition, the output of our proposed transpiler is a high-level program that satisfies the constraints of a target that can have a different architecture from the source.

Expressing and verifying parser behavior. The principles of parser design were discussed by Gibb

et al. [51]. We can consider a parser as a finite state machine (FSM). The state transition logic can be determined by comparing certain fields against constants and each parser node executes packet extractions to identify packet headers from the input bit stream. Leapfrog [41] proposes a new framework to verify the equivalence of 2 parser structures. However, our goal is to generate a new parser through a synthesis procedure, which is orthogonal to their implementation.

5.6 Summary

The proliferation of programmable network devices motivates engineers to write programs for different targets. This paper proposes to build a program synthesis-based transpiler, Polyglotter, that realizes transpilation for the parser part of P4 and NPL programs over Tofino and Trident 4 programmable switches. Our initial results show the correctness and efficiency of Polyglotter. We hope our work can prompt further research on the transpiler design across more network languages and hardware devices.

6 Conclusion

To conclude this dissertation, We briefly summarize our work, mention its potential impact on more applications, and outline several future directions.

6.1 Beyond our work

This thesis starts by using program synthesis to do code generation for programmable switches, then combines more solver techniques (e.g., ILP solver, SMT solver) to do code generation and resource allocation for the whole packet processing pipeline, and extends the idea of solver-based compiler design to the parser portion.

Experiment results show that compared with traditional compilers that design a series of program rewriting rules, solver-aided compilers can output better compilation results in terms of hardware resource usage within a reasonable time. Our developed speed-up algorithm can reduce the compilation time a lot (e.g., from hours to minutes) by dividing one big problem into multiple sub-problems.

This thesis did an initial trial to incorporate solver-based techniques into the design of optimizing compilers for programmable network devices. We also believe that we can extend solver techniques into broader domains, including network verification, programming language design, network function offloading, and compiler design for more hardware devices. Although there may be some limitations in terms of resource constraint encoding and running time of the solver,

it is convincing to us that solvers can help improve the general efficiency of many system-related operations to some extent.

6.2 Future Directions

Several directions based on this thesis are worth further exploration for future work.

6.2.1 Code generation for line-rate parsers

Polyglotter realizes the transpilation over several simple parser benchmarks. It, first of all, does transpilation in the granularity of parser nodes to capture the state transition logic and then synthesizes the packet extraction behavior. A natural extension is to synthesize a parser as a whole without such a separation. It is possible to frame that as a finite state transducer synthesis [53] problem.

Additionally, we find that it is possible to implement the behavior of a parser in the packet-processing pipeline. As an example, we could “store" the input bitstream in predefined temporary variables in the parser and assign values to packet fields through match-action tables in the pipeline. We put state transition keys in the parser as keys of the match action table while updating packet fields in action by using bitwise operations. Similarly, some simple match-action tables can be placed in the parser (e.g., logical table operations between parse break and parse continue).

Research Questions. Given we have the flexibility to execute the parser’s behavior in the pipeline and vice versa, we want to realize it automatically by fully utilizing all available hardware resources. This leads to multiple interesting questions. How do we efficiently encode the target device’s parser behavior model and its resource constraints into a synthesis engine (e.g., SKETCH, z3, Rosette)? How to leverage customized speed-up algorithms (e.g., CEGIS loop) to complete such a process faster? Could we provide a uniform design for both the pipeline and parser so that

the transpiler could determine which part to run in the packet-processing pipeline and which part to run in the parser?

6.2.2 Approximate Program Synthesis

There are many situations where data plane resources are heavily constrained, necessitating approximate—but fast—packet processing. Examples include sampled statistics, measurement sketches that trade-off counter accuracy for line-rate performance and reduced memory, and multi-tenant scenarios where it is essential to pack as many network programs as possible into the switch or NIC [97]. Each such situation today requires developing a custom approach to trade accuracy for resource savings or high performance.

Program synthesis can provide a general method to reduce program resource usage through approximation. Approximate compilers [43, 96, 121, 126] already exist to target hardware with instructions that reduce energy. Recently, a more general *approximate program synthesis* framework [28, 29] has emerged. This framework has been used to improve the performance of some programs by an order of magnitude [28] while producing approximate results with bounded errors.

Research Questions. Network programs are typically written as functions over a single packet, e.g., in P4. How should one synthesize network programs with bounded inaccuracy over a packet *trace*, rather than just a single packet? How should we address resource constraints like memory usage, which depends on the workload (e.g., number of flows) and not just the program? Given a library of high-performance primitives such as counting, hashing, etc., is it possible to synthesize measurement sketches (e.g., count-min sketch) that capture a statistic with guaranteed memory-accuracy tradeoffs?

6.2.3 Synthesizing Program Repairs

Developers of packet processing programs frequently need to troubleshoot correctness, security, and performance issues with their software. While it is impossible to remove the need for human insight from troubleshooting, we believe it is beneficial to generate human-interpretable repair hints automatically. Previous work includes automatically generating feedback on grading [127] using program synthesis. Yet, there are still quite a few avenues today to provide such hints to ease the troubleshooting process.

Small, localized rewrites of the program source code can serve as useful hints to fix many issues. Examples include suggesting edits to a program to fit it into a switch pipeline, rewrites for offending eBPF program code to move past eBPF verification errors [15, 143], and hints to rewrite "hot" code regions of a DPDK program to improve its performance.

Research Questions. Is it possible to generate local rewrites to fit a problematic network program into a packet-processing pipeline? Can we speed up a slow network program by replacing hot code regions with fast implementations using a database of localized code rewrites [21]? Can program synthesis replace unsafe data flows in an eBPF program with safe ones without drastically changing the whole program? It may often be necessary to change the semantics of a program to fix an issue. Can we develop a domain-specific measure of the semantic distance between the rewritten program and the original one? How should a synthesizer use such a measure when suggesting rewrites?

6.2.4 Speeding up the program synthesis using the Large Language Model (LLM)

One of the major bottlenecks of using program synthesis is its long running time. For instance, in both Chipmunk [50] and CaT [48] compilers, we optimize the synthesis procedure using domain-specific algorithms (e.g., slicing and division between stateful/stateless computa-

tion). Previous work includes doing parallel searching [70] using multi-core. Recently emerging LLM techniques provide another option to speed up the whole process. Despite machine learning tools such as LLM can be scalable in many situations, this is at the cost of accuracy loss (failure to achieve 100% accuracy), making it difficult to play a role in compiler design. I want to explore the potential synergy effect in between.

Research Questions. Can we speed up the program synthesis engine using LLM? If we can leverage LLM techniques to prioritize the program synthesis engine's solution-searching strategy, it is possible to extend the usage of program synthesis into more application scenarios. The heuristic behind it originated from my experience of running program synthesis in Chipmunk and CaT compilers, where the compilation time would be improved significantly if we could manually determine a tiny portion of the whole search space using human intelligence. There is a possibility that LLM can help us detect some useful features that cannot be found by human beings. All synthesis benchmarks from this thesis can be used as model training data.

6.2.5 Implementing abstract data structures into programmable network devices

We can regard programmable network devices as not only data computation units but also data processing ones. As a result, they can do data collection and leave the data processing work to attached CPU cores. Initial trials include a proposal for state-compute replication (SCR) in [153]. This runs a packet history recorder over NIC or switches to realize state updates across multiple cores without synchronization in between. The ring buffer data structure is used in SCR to record the history of a packet sequence. In addition, I have performed some preliminary tests on several data structures, such as queue, array, and stack.

Research Questions. Can we correctly implement abstract data structures with the help of human intelligence? I hope to do further research in 2 main directions based on the experience

of both data structure and ALU functionality of various network devices: (1) exploring the pool of abstract data structures that can be executed over the existing network devices and (2) offering formal proof checker to verify the correctness of the manual data structure implementation.

Bibliography

[1] *[Sketchusers] Strange error for large integer constants*. https://lists.csail.mit.edu/pipermail/sketchusers/2019-July/000094.html.

[2] *A machine model for line-rate programmable switches*. https://github.com/packet-transactions/banzai.

[3] Alessandro Abate et al. "Counterexample Guided Inductive Synthesis Modulo Theories". In: *Computer Aided Verification*. 2018.

[4] *Actions, P4-16 specification v1.2.3*. https://p4.org/wp-content/uploads/2022/07/P4-16-spec.html#sec-table-action-list. 2022.

[5] Anup Agarwal et al. "Towards provably performant congestion control". In: *USENIX NSDI*. 2024, pp. 951–978.

[6] Maaz Bin Safeer Ahmad and Alvin Cheung. "Automatically Leveraging MapReduce Frameworks for Data-Intensive Applications". In: *ACM SIGMOD*. 2018.

[7] Alfred V. Aho, Stephen C. Johnson, and Jeffrey D. Ullman. "Code Generation for Expressions with Common Subexpressions". In: *Journal of the ACM* (1977).

[8] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811.

[9] V Aho Alfred, S Lam Monica, and D Ullman Jeffrey. *Compilers Principles, Techniques & Tools*. pearson Education, 2007.

[10] Mohammad Alizadeh et al. “CONGA: Distributed Congestion-aware Load Balancing for Datacenters”. In: *ACM SIGCOMM*. 2014.

[11] Jesus M. Almendros-Jimenez, Josep Silva, and Salvador Tamarit. “XQuery Optimization Based on Program Slicing”. In: *CIKM*. 2011.

[12] R. Alur et al. “Syntax-guided synthesis”. In: *2013 Formal Methods in Computer-Aided Design*. 2013.

[13] *Amazon EC2 Spot Instances Pricing*. https://aws.amazon.com/ec2/spot/pricing/.

[14] *AMD Pensando Infrastructure Accelerators*. https://www.amd.com/en/accelerators/pensando.

[15] *An eBPF overview, part 1: Introduction*. https://www.collabora.com/news-and-blog/blog/2019/04/05/an-ebpf-overview-part-1-introduction/.

[16] Carolyn Jane Anderson et al. “NetKAT: semantic foundations for networks”. In: POPL ’14. San Diego, California, USA: Association for Computing Machinery, 2014, pp. 113–126. ISBN: 9781450325448. DOI: 10.1145/2535838.2535862.

[17] *Announcing Amazon EC2 per second billing*. https://aws.amazon.com/about-aws/whats-new/2017/10/announcing-amazon-ec2-per-second-billing/.

[18] Mina Tahmasbi Arashloo, Ryan Beckett, and Rachit Agarwal. “Formal Methods for Network Performance Analysis”. In: *USENIX NSDI*. 2023.

[19] Mina Tahmasbi Arashloo et al. “SNAP: Stateful network-wide abstractions for packet processing”. In: *ACM SIGCOMM*. 2016.

[20] Jean-Philippe Aumasson and Daniel J. Bernstein. *SipHash: a fast short-input PRF*. Cryptology ePrint Archive, Paper 2012/351. https://eprint.iacr.org/2012/351. 2012.

[21] Sorav Bansal and Alex Aiken. “Automatic Generation of Peephole Superoptimizers”. In: *ACM ASPLOS*. 2006.

[22] Clark W. Barrett et al. “Satisfiability Modulo Theories”. In: *Handbook of Satisfiability*.

[23] Ran Ben Basat et al. “PINT: Probabilistic In-Band Network Telemetry”. In: *ACM SIGCOMM*. 2020.

[24] *Benchmarks used to show resource transformation difference*. https://anonymous.4open.science/r/program_transformation_ex-6EE7.

[25] Dirk Beyer, Sumit Gulwani, and David A. Schmidt. “Combining Model Checking and Data-Flow Analysis”. In: *Handbook of Model Checking*. Springer, 2018.

[26] Leyla Bilge et al. “EXPOSURE: Finding Malicious Domains Using Passive DNS Analysis”. In: *USENIX NDSS*. 2011.

[27] Rudiger Birkner et al. “Config2Spec: Mining Network Specifications from Network Configurations”. In: *USENIX NSDI*. 2020.

[28] James Bornholt et al. “Approximate Program Synthesis”. In: *Workshop on Approximate Computing Across the Stack*. 2015.

[29] James Bornholt et al. “Optimizing Synthesis with Metasketches”. In: *ACM POPL*. 2016.

[30] Pat Bosshart et al. “Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN”. In: *ACM SIGCOMM*. 2013.

[31] Pat Bosshart et al. “P4: Programming Protocol-independent Packet Processors”. In: *SIGCOMM CCR* (2014).

[32] *Broadcom Jericho2 Ethernet Switch Series*. https://www.broadcom.com/products/ethernet-connectivity/switching/stratadnx/bcm88690.

[33] *Broadcom Trident Ethernet Switch Series*. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series.

[34] Sharad Chole et al. “dRMT: Disaggregated Programmable Switching”. In: *ACM SIGCOMM*. 2017.

[35] *Concurrency Model for P4*. https://github.com/p4lang/p4-spec/issues/48.

[36] J. Cong and Yuzheng Ding. "FlowMap: an optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (1994).

[37] Philippe Coussy et al. "An Introduction to High-Level Synthesis". In: *IEEE Design Test of Computers* (2009).

[38] Ron Cytron et al. "Efficiently computing static single assignment form and the control dependence graph". In: *ACM TOPLAS* (1991).

[39] Jack W. Davidson and Christopher W. Fraser. "The Design and Application of a Retargetable Peephole Optimizer". In: *TOPLAS* (1980).

[40] *DepQBF Solver*. http://lonsing.github.io/depqbf/.

[41] Ryan Doenges et al. "Leapfrog: certified equivalence for protocol parsers". In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 950–965. ISBN: 9781450392655. DOI: 10.1145/3519939.3523715.

[42] *Domino Compiler*. https://github.com/chipmunk-project/domino-compiler.

[43] Hadi Esmaeilzadeh et al. "Neural Acceleration for General-Purpose Approximate Programs". In: *IEEE MICRO*. 2012.

[44] Grigory Fedyukovich, Maaz Bin Safeer Ahmad, and Rastislav Bodik. "Gradual Synthesis for Static Parallelization of Single-pass Array-processing Programs". In: *ACM PLDI*. 2017.

[45] Wu-chang Feng et al. "The BLUE Active Queue Management Algorithms". In: *IEEE/ACM Transactions on Networking* (2002).

[46] Jiaqi Gao et al. "Lyra: A Cross-Platform Language and Compiler for Data Plane Programming on Heterogeneous ASICs". In: *ACM SIGCOMM*. 2020.

[47] Xiangyu Gao et al. “Autogenerating Fast Packet-Processing Code Using Program Synthesis”. In: *ACM HotNets*. 2019.

[48] Xiangyu Gao et al. “CaT: A Solver-Aided Compiler for Packet-Processing Pipelines”. In: *ACM ASPLOS*. Vancouver, BC, Canada, 2023.

[49] Xiangyu Gao et al. “Cross-Platform Transpilation of Packet-Processing Programs using Program Synthesis ”. In: *ACM APNET*. 2024.

[50] Xiangyu Gao et al. “Switch Code Generation Using Program Synthesis”. In: *ACM SIGCOMM*. 2020.

[51] Glen Gibb et al. “Design principles for packet parsers”. In: *Architectures for Networking and Communications Systems*. IEEE. 2013.

[52] Pawan Goyal, Harrick M. Vin, and Haichen Chen. “Start-time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks”. In: *ACM SIGCOMM*. 1996.

[53] Anvay Grover, Ruediger Ehlers, and Loris D’Antoni. “Synthesizing Transducers from Complex Specifications.” In: *FMCAD*. 2022, pp. 294–303.

[54] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. “Program Synthesis”. In: *Foundations and Trends® in Programming Languages* 4 (2017).

[55] Sumit Gulwani et al. “Synthesis of Loop-Free Programs”. In: *ACM PLDI*. 2011.

[56] *Gurobi Optimizer*. https://www.gurobi.com/.

[57] Gary D. Hachtel and Fabio Somenzi. *Logic synthesis and verification algorithms*. Springer, 2006.

[58] Ahmed El-Hassany et al. “NetComplete: Practical Network-Wide Configuration Synthesis with Autocompletion”. In: *USENIX NSDI*. 2018.

[59] *High-Capacity StrataXGS Trident 4 Ethernet Switch Series*. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series.

[60] Mary Hogan et al. "Modular Switch Programming Under Resource Constraints". In: *USENIX NSDI*. 2022.

[61] *HPCC++: Enhanced High Precision Congestion Control*. https://www.ietf.org/id/draft-miao-tsv-hpcc-01.html.

[62] Qinheping Hu et al. "Proving Unrealizability for Syntax-Guided Synthesis". In: *CAV*. 2019.

[63] Qun Huang et al. "SketchVisor: Robust Network Measurement for Software Packet Processing". In: *ACM SIGCOMM*. 2017.

[64] Stephen Ibanez, Muhammad Shahbaz, and Nick McKeown. "The Case for a Network Fast Path to the CPU". In: *ACM HotNets*. 2019.

[65] Stephen Ibanez et al. "The P4->NetFPGA Workflow for Line-Rate Packet Processing". In: *ACM/SIGDA FPGA*. 2019.

[66] *Intel FlexPipe*. http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ethernet-switch-fm6000-series-brief.pdf.

[67] *Intel FPGA's High Level Synthesis Compiler*. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html.

[68] *Intel Tofino Programmable Ethernet Switch ASIC*. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html.

[69] *Intel® Infrastructure Processing Unit (Intel® IPU)*. https://www.intel.com/content/www/us/en/products/details/network-io/ipu.html.

[70] Jinseong Jeon et al. "Adaptive Concretization for Parallel Program Synthesis". In: *CAV*. 2015.

[71] Xin Jin et al. “NetCache: Balancing Key-Value Stores with Fast In-Network Caching”. In: *ACM SOSP*. 2017.

[72] Xin Jin et al. “NetChain: Scale-Free Sub-RTT Coordination”. In: *USENIX NSDI*. 2018.

[73] Yuwei Jin et al. “Quantum Fourier Transformation Circuits Compilation”. In: *arXiv preprint arXiv:2312.16114* (2023).

[74] *Jinja - Jinja Documentation (2.10.x)*. https://jinja.palletsprojects.com/en/2.10.x/.

[75] Lavanya Jose et al. “Compiling Packet Programs to Reconfigurable Switches”. In: *USENIX NSDI*. 2015.

[76] Shoaib Kamil et al. “Verified Lifting of Stencil Computations”. In: *ACM SIGPLAN*. 2016.

[77] Anirudh Sivaraman Kaushalram. “Designing Fast and Programmable Routers”. PhD thesis. EECS Department, Massachusetts Institute of Technology, Sept. 2017.

[78] Changhoon Kim et al. “In-band network telemetry via programmable dataplanes”. In: *SIGCOMM Industrial Demo Session*. 2015.

[79] James C. King. “Symbolic Execution and Program Testing”. In: *Communications of the ACM* (1976).

[80] Christos Kozanitis et al. “Leaping Multiple Headers in a Single Bound: Wire-Speed Parsing Using the Kangaroo System”. In: *2010 Proceedings IEEE INFOCOM*. 2010, pp. 1–9. DOI: 10.1109/INFCOM.2010.5462139.

[81] Taek Jin Kwon and Mario Gerla. “Efficient flooding with passive clustering (PC) in ad hoc networks”. In: *ACM SIGCOMM Computer Communication Review* 32.1 (2002), pp. 44–56.

[82] U.S. Bureau of Labor Statistics. *Occupational Employment and Wages, May 2018, Software developers, Systems software*. https://www.bls.gov/oes/current/oes151133.htm.

[83] M. Lam. “Software Pipelining: An Effective Scheduling Technique for VLIW Machines”. In: *ACM PLDI*. 1988.

[84] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *ACM/IEEE CGO*. 2004.

[85] *lcc, A Retargetable Compiler for ANSI C*. https://drh.github.io/lcc/.

[86] Yifan Li et al. “Cetus: Releasing P4 Programmers from the Chore of Trial and Error Compiling”. In: *USENIX NSDI*. 2022.

[87] Yuliang Li et al. “HPCC: High Precision Congestion Control”. In: *ACM SIGCOMM*. 2019.

[88] *LLVM Link Time Optimization: Design and Implementation*. https://llvm.org/docs/LinkTimeOptimization.html.

[89] Robert MacDavid, Xiaoqi Chen, and Jennifer Rexford. “Scalable Real-Time Bandwidth Fairness in Switches”. In: *IEEE INFOCOM*. 2023.

[90] *MarkusRabe/cadet: A fast and certifying solver for quantified Boolean formulas*. https://github.com/MarkusRabe/cadet.

[91] Roland Meier et al. “NetHide: Secure and Practical Network Topology Obfuscation”. In: *USENIX Security*. 2018.

[92] *Mellanox Innova-2 Flex Open Programmable SmartNIC*. https://www.mellanox.com/page/products_dyn?product_family=276&mtag=programmable_adapter_cards_innova2flex&ssn=7j4vr3u5elh91qnkb9ubjsdlo4.

[93] *Mellanox Products: Spectrum 2 Ethernet Switch ASIC*. https://www.mellanox.com/page/products_dyn?product_family=277&mtag=spectrum2_ic.

[94] *Menshen: An Isolation Mechanism for High-Speed Packet-Processing Pipelines*. https://isolation.quest/.

[95] Rui Miao et al. “SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs”. In: *ACM SIGCOMM*. 2017.

[96] Sasa Misailovic et al. “Chisel: Reliability- and Accuracy-aware Optimization of Approximate Computational Kernels”. In: *ACM OOPSLA*. 2014.

[97] Masoud Moshref et al. “DREAM: Dynamic Resource Allocation for Software-defined Measurement”. In: *ACM SIGCOMM*. 2014.

[98] Srinivas Narayana et al. “Language-Directed Hardware Design for Network Performance Monitoring”. In: *ACM SIGCOMM*. 2017.

[99] *NPL Specification*. https://github.com/nplang/NPL-Spec.

[100] *P4 Compiler*. https://github.com/p4lang/p4c.

[101] *P4 Extern Types*. https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html#sec_extern.

[102] *P4 Studio | Barefoot*. https://www.barefootnetworks.com/products/brief-p4-studio/.

[103] *P4 Tutorial*. https://github.com/p4lang/tutorials.

[104] *P4-16 language specification*. https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.html.

[105] *P4_16 language specification*. https://p4lang.github.io/p4-spec/docs/P4-16-v1.2.2.html#sec-concurrency.

[106] *p4lang/switch: Consolidated switch repo (API, SAI and Nettlink)*. https://github.com/p4lang/switch.

[107] Tian Pan et al. “Sailfish: Accelerating Cloud-Scale Multi-Tenant Multi-Service Gateways with Programmable Switches”. In: *ACM SIGCOMM*. 2021.

[108] Aurojit Panda et al. “Verifying Reachability in Networks with Mutable Datapaths”. In: *USENIX NSDI*. 2017.

[109] *Pensando Distributed Services Architecture SmartNIC*. http://www.servethehome.com/pensando-distributed-services-architecture-smartnic/.

[110] Phitchaya Mangpo Phothilimthana et al. “Chlorophyll: Synthesis-aided Compiler for Low-power Spatial Architectures”. In: *ACM PLDI*. 2014.

[111] Phitchaya Mangpo Phothilimthana et al. “Scaling Up Superoptimization”. In: *ACM ASPLOS*. 2016.

[112] Phitchaya Mangpo Phothilimthana et al. “Swizzle Inventor: Data Movement Synthesis for GPU Kernels”. In: *ACM ASPLOS*. 2019.

[113] Gordon D. Plotkin et al. “Scaling Network Verification Using Symmetry and Surgery”. In: *ACM POPL*. 2016.

[114] *Product Brief Tofino Page | Barefoot*. https://barefootnetworks.com/products/brief-tofino/.

[115] *Programming the Forwarding Plane - Nick McKeown*. https://forum.stanford.edu/events/2016/slides/plenary/Nick.pdf.

[116] *Quantified Boolean Formula*. https://en.wikipedia.org/wiki/True_quantified_Boolean_formula.

[117] *Rate enforcer P4 program github repo*. https://github.com/Princeton-Cabernet/AHAB/blob/7c3b1bb/p4src/include/rate_enforcer.p4#L366.

[118] Regehr, John. *Synthesizing Constants*. https://blog.regehr.org/archives/1636.

[119] Keith W Ross and James F Kurose. *Computer networking*. 2012.

[120] Shambwaditya Saha, Santhosh Prabhu, and P. Madhusudan. “NetGen: Synthesizing Data-plane Configurations for Network Policies”. In: *ACM SOSR*. 2015.

[121] Adrian Sampson et al. “EnerJ: Approximate Data Types for Safe and General Low-power Computation”. In: *ACM PLDI*. 2011.

[122] Amedeo Sapio et al. “In-Network Computation is a Dumb Idea Whose Time Has Come”. In: *ACM HotNets*. 2017.

[123] Eric Schkufza, Rahul Sharma, and Alex Aiken. “Stochastic Superoptimization”. In: *ACM ASPLOS*. 2013.

[124] Muhammad Shahbaz and Nick Feamster. “The case for an intermediate representation for programmable data planes”. In: *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. SOSR ’15. Santa Clara, California: Association for Computing Machinery, 2015. ISBN: 9781450334518. DOI: 10.1145/2774993.2775000.

[125] Bhavana Vannarth Shobhana, Srinivas Narayana, and Badri Nath. “Load balancers need in-band feedback control”. In: *ACM HotNets*. 2022.

[126] Stelios Sidiroglou-Douskos et al. “Managing Performance vs. Accuracy Trade-offs with Loop Perforation”. In: *ESEC/FSE*. 2011.

[127] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. “Automated Feedback Generation for Introductory Programming Assignments”. In: *ACM PLDI*. 2013.

[128] Shan Sinha, Srikanth Kandula, and Dina Katabi. “Harnessing TCPs Burstiness using Flowlet Switching”. In: *HotNets*. 2004.

[129] *SipHash P4 program github repo*. https://github.com/Princeton-Cabernet/p4-projects/blob/master/SipHash-tofino/p4src/siphash24_ingressonly.p4#L587.

[130] Anirudh Sivaraman et al. “Packet Transactions: High-Level Programming for Line-Rate Switches”. In: *ACM SIGCOMM*. 2016.

[131] *Sketch Source Code*. https://people.csail.mit.edu/asolar/sketch-1.7.5.tar.gz.

[132] Armando Solar Lezama. “Program Synthesis By Sketching”. PhD thesis. EECS Department, University of California, Berkeley, 2008.

[133] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. “Sketching Concurrent Data Structures”. In: *ACM PLDI*. 2008.

[134] Armando Solar-Lezama et al. “Combinatorial Sketching for Finite Programs”. In: *ACM ASPLOS*. 2006.

[135] John Sonchak et al. “Lucid: A Language for Control in the Data Plane”. In: *ACM SIGCOMM*. 2021.

[136] Haoyu Song. “Protocol-oblivious Forwarding: Unleash the Power of SDN Through a Future-proof Forwarding Plane”. In: *ACM HotSDN*. 2013.

[137] Hardik Soni et al. “Composing dataplane programs with μP4”. In: *ACM SIGCOMM*. 2020.

[138] Kausik Subramanian, Loris D’Antoni, and Aditya Akella. “Genesis: Synthesizing Forwarding Tables in Multi-tenant Networks”. In: *ACM POPL*. 2017.

[139] Kausik Subramanian, Loris D’Antoni, and Aditya Akella. “Synthesis of Fault-Tolerant Distributed Router Configurations”. In: *ACM SIGMETRICS*. 2018.

[140] Nik Sultana et al. “Flightplan: Dataplane disaggregation and placement for p4 programs”. In: *USENIX NSDI*. 2021.

[141] *Synopsys Introduces Synphony High Level Synthesis*. https://news.synopsys.com/index.php?s=20295&item=123096.

[142] C.H. Tai, J. Zhu, and N. Dukkipati. “Making Large Scale Deployment of RCP Practical for Real Networks”. In: *INFOCOM*. 2008.

[143] *The art of writing eBPF programs: a primer.|Sysdig*. https://sysdig.com/blog/the-art-of-writing-ebpf-programs-a-primer/.

[144] *The P4 Language Specification*. https://p4.org/p4-spec/p4-14/v1.0.5/tex/p4.pdf.

[145] *The reference P4 software switch*. https://github.com/p4lang/behavioral-model.

[146] *The Z3 Theorem Prover*. https://github.com/Z3Prover/z3.

[147] Emina Torlak and Rastislav Bodik. "Growing Solver-aided Languages with Rosette". In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. 2013.

[148] *Trident 4 / BCM56690 Series*. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56690.

[149] Balázs Vass et al. "Compiling Packet Programs to Reconfigurable Switches: Theory and Algorithms". In: *EuroP4*. 2020.

[150] Tao Wang et al. "Isolation Mechanisms for High-Speed Packet-Processing Pipelines". In: *USENIX NSDI*. 2022.

[151] Mark Weiser. "Program Slicing". In: *ICSE*. 1981.

[152] Michael D. Wong, Aatish Kishan Varma, and Anirudh Sivaraman. "Testing Compilers for Programmable Switches Through Switch Hardware Simulation". In: *ACM CoNEXT*. 2020.

[153] Qiongwen Xu et al. "State-Compute Replication: Parallelizing High-Speed Stateful Packet Processing". In: *USENIX NSDI*. 2025.

[154] Qiongwen Xu et al. "Synthesizing safe and efficient kernel extensions for packet processing". In: *ACM SIGCOMM*. 2021.

[155] Sophia Yoo and Xiaoqi Chen. "Secure Keyed Hashing on Programmable Switches". In: *ACM SIGCOMM Workshop on Secure Programmable Network INfrastructure*. 2021.

[156] Lihua Yuan, Chen-Nee Chuah, and Prasant Mohapatra. "ProgME: towards programmable network measurement". In: *ACM SIGCOMM*. 2007.